Aplicando refatorações em uma arquitetura de software monolítica para uma solução utilizando microsserviços

Alan Ricardo S. Silval[1], Ana Claudia Rossi[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie (UPM)
01.302-907 – São Paulo – SP – Brasil

alan.ricardo.silva1@gmail.com, anaclaudia.rossi@mackenzie.br

Abstract. With the constant growth of current systems and with the development of the complexity required by them, the monolithic architecture in systems with a very high growth potential becomes an unfeasible solution. The microservice software architecture has brought great results in terms of scalability, maintenance and security, making it a more coherent solution for current systems. This research aims to apply a refactoring pattern in a generic monolithic architecture to a model that uses microservices

Resumo. *Com o crescimento constante dos sistemas atuais e com o desenvolvimento da complexidade exigida por eles, a arquitetura monolítica em sistemas com um potencial de crescimento muito grande torna-se uma solução inviável. A arquitetura de software de microsserviços vem trazendo grandes resultados no quesito escalabilidade, manutenção e segurança, tornando-a uma solução mais coerente para os sistemas atuais. Esta pesquisa tem como objetivo aplicar um padrão de refatoração em uma arquitetura monolítica genérica para um modelo que utiliza microsserviços*

1. Introdução

Ao longo dos anos o cenário de desenvolvimento vem em uma crescente e com isso vem surgindo novas tecnologias, como linguagens de programação, paradigmas e processos. Sendo assim, para manter o desempenho e a produção, as empresas estão se modernizando principalmente na área da arquitetura de software (Di Francesco, 2017).

Os monólitos são um tipo de sistema onde todos os módulos importantes de uma aplicação são agrupados em uma única peça e todas as funcionalidades são implementadas e executadas em conjunto (Newman, 2019).

As aplicações que usam o sistema de arquitetura monolítica são planejadas para serem autocontidas e os componentes da aplicação são evidentes e interdependentes. No caso de uma mesma mudança de função, devido à alta dependência do aplicativo, pode interferir em outras funções (Rouse, 2016).

No desenvolvimento de sistemas de software, a arquitetura de microsserviços é uma alternativa à arquitetura monolítica, o objetivo é melhorar o sistema e resolver alguns problemas, principalmente os problemas que surgem durante o processo de atualização (Bucshmann, 1996).

As principais vantagens de refatorar um monólito para arquitetura de microsserviços são: implementação independente, fácil escalabilidade da aplicação, fácil manutenção e liberdade para trabalhar com mais de uma tecnologia (Viggiato, 2018).

Sabendo disso, essa pesquisa tem como intuito apresentar o resultado de uma refatoração de uma arquitetura monolítica para uma arquitetura utilizando microsserviços utilizando como base modelo genérico que permite o planejamento de atividades de um recurso em um plano semestral (plano de atividades do recurso, PAR).

2. Metodologia

Visando o nível de complexidade e a dimensão que envolve o processo de modelagem e refatoração de um projeto existente e disponível em tempo real, dando a necessidade de realizar a pesquisa em ambiente de sua aplicação, a metodologia adotada para a condução desta pesquisa foi a Design Science Research. Essa escolha se deu pela oportunidade real de identificar um problema real em um ambiente de software, que utiliza uma aplicação que adotada a arquitetura monolítica trazendo uma série de problemas como disponibilidade, escalabilidade, dificuldade em realizar manutenções e limitando os desenvolvedores a se manter a uma única tecnologia, e aplicando a solução de migrar a aplicação para uma arquitetura que adota os microsserviços como padrão de desenvolvimento de software.

Na Figura 1, é ilustrado o modelo de desenvolvimento de pesquisa inspirado no Design Science Research que é dividido em 3 etapas. A primeira etapa trata-se de identificar um objetivo e uma motivação para a realização da pesquisa. Definir uma pesquisa específica e justificar o problema dando valor a uma possível solução. A definição do problema será utilizada para desenvolver os artefatos que possivelmente serão utilizados para fornecer uma solução. A segunda etapa trata-se de definir os objetivos e uma solução. Inferir os objetivos de uma solução a partir da definição do problema e do conhecimento do que é possível e viável. A terceira etapa refere-se a projetar um design e desenvolver a solução. Nesta etapa é onde são criados os artefatos. Os artefatos são potencialmente construções, modelos, métodos ou instâncias. Conceitualmente, um artefato de pesquisa em design pode ser definido como qualquer objeto projetado como uma contribuição de pesquisa que esteja incorporada ao design. Sendo assim a atividade tem como objetivo determinar a funcionalidade do artefato e sua arquitetura, e em seguida, criar um artefato real.

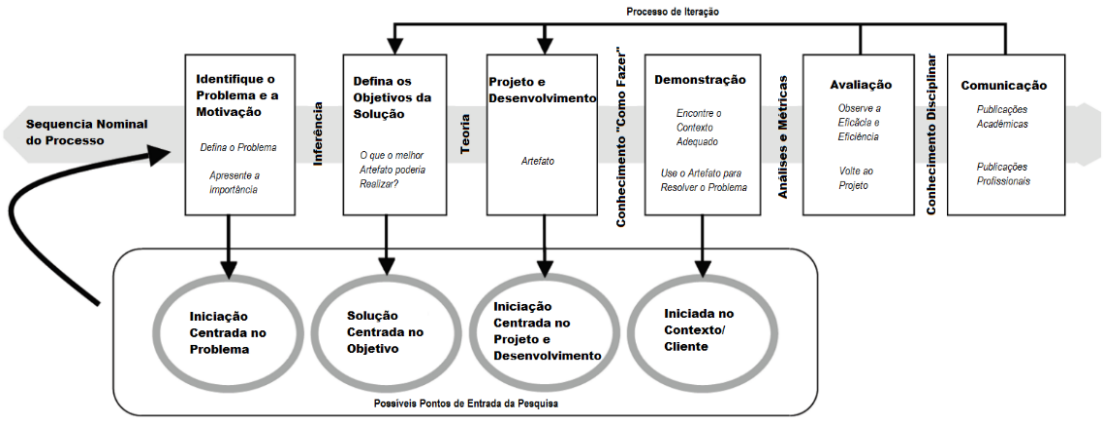

Figura 1. Modelo de Processo DSRM (Design Science Research Methodology)

Fonte: A Design Science Research Methodology for Information Systems Research

3. Objetivo

O objetivo geral deste trabalho é apresentar um resultado de refatoração de uma modelo de software com arquitetura monolítica para uma que adota uma arquitetura utilizando microsserviços. Esse processo foi aplicado no contexto de migração do sistema PAR. Para alcançar o objetivo geral desta pesquisa apresenta-se os seguintes objetivos específicos:

1. Extrair, formular e conduzir o processo de refatoração de um modelo de software que possui uma arquitetura monolítica para uma que adota a arquitetura de microsserviços.
2. Avaliar os impactos da refatoração do modelo.

4. Arquitetura de Software

Ao longo dos anos nota-se que junto ao aumento da complexidade e dimensão de sistema de informação, bem como ocorreu o aumento da complexidade em organizar e estruturar componentes de softwares (SORDI, MARINHO & NAGY, 2006). A arquitetura de software é definida como o esqueleto do software, então, torna-se o plano de mais alto-nível da construção de novos sistemas. Na arquitetura de software o planejamento define que a estrutura lide melhor com o sistema idealizado (KRAFZIG, BANKE e SLAMA, 2004).

Por consequência, existem vários gêneros (domínios) de arquitetura de software, sendo cada um deles apropriados de acordo com o tipo da aplicação (PRESSMAN, 2011). Sendo assim, é necessário identificar e compreender a arquitetura de software, visto que é a base do sistema (SOMMERVILLE, 2007).

Em vista disso, Pressman cita que a arquitetura de software representa uma estrutura onde um conjunto de entidades (apresentados como componentes) interligado a um grupo de relacionamentos determinados (denominados conectores) (PRESSMAN, 2011).

Já para Sommerville, a arquitetura de software é um framework essencial para estruturar o sistema, levando os requisitos de qualidade. Essa influência acontece devido às resoluções tomadas na arquitetura do projeto que incluem a escolha do tipo de aplicação, a distribuição do sistema, os estilos arquitetônicos a serem usados e as medidas que a arquitetura deve ser documentada e avaliada (SOMMERVILLE, 2007).

4.1 Arquitetura Monolítica

A arquitetura monolítica é a mais utilizada em softwares corporativos. Sua existência ganhou valor no início das primeiras implantações de software para Web, sendo a responsável por manter todos os componentes do lado do servidor em uma única unidade (RICHARDSON, 2014).

Na arquitetura monolítica as execuções do processo são feitas em apenas um processo, tendo elas um único processo. Sendo assim, ao usar os recursos mais básicos de sua linguagem, até mesmo testar e executar o aplicativo no laptop do próprio desenvolvedor (Fowler, 2014).

Segundo Richardson existe uma facilidade de desenvolver aplicações monolíticas, tendo em vista as IDES e as demais ferramentas de desenvolvimento são orientadas para o desenvolvimento de um único aplicativo. Afirma ele que para aplicações de pequeno porte a arquitetura monolítica se sobressai por ser mais fácil de testar e fácil de implantar (RICHARDSON, 2014).

Infelizmente as aplicações monolíticas começam a se tornar um problema de acordo com o seu nível de crescimento e complexidade, tendo em vista que para realizar modificações e entender suas funcionalidades quando a equipe de desenvolvimento sofre mudanças (NAMIOT, 2014).

4.2 Arquitetura de Microsserviços

A arquitetura de microsserviços é uma maneira de projetar aplicativos onde se utilizam suítes de serviços tornando o sistema independente (Fowler, 2014). Essa arquitetura realiza a construção de aplicações onde seus serviços são implantados individualmente e independentemente de maneira escalável. Sendo assim, a possibilidade de escrita de serviços em diferentes tipos de linguagens de programação torna-se uma vantagem. A arquitetura é uma aproximação para desenvolver um único aplicativo como uma série de pequenos serviços, sendo eles executados individualmente. Esses serviços são implementados de maneira independente e com implantação automatizada (Fowler, 2014).

Newman diz que os microsserviços são autônomos e independentes, com tamanho reduzido, sendo cada um deles responsável por ter tarefas bem definidas e podendo trabalhar em conjunto, mas não possuem interdependência. Isso apresenta uma liberdade em relação à decisão na hora de selecionar uma tecnologia e no ganho de velocidade na implementação para novas funcionalidades com o propósito de atender às novas demandas de negócio, colocando as necessidades reais da empresa na economia atual (NEWMAN, 2015).

Na fase de especificação do design arquitetural, a escolha do número e o tamanho dos microsserviços é realizada de forma individual, tornando o conhecimento do domínio e a arquitetura necessários. O que leva um serviço com diferentes funcionalidades não associadas pode precisar ser dividido para melhorar sua estabilidade, manutenção e implantação (DAYA, 2016).

De forma destoante aos sistemas monolíticos, os microsserviços recebem suas responsabilidades de forma planejada, restrita e escalável, podendo também ser possível a utilização de recursos computacionais e de menor capacidade e melhor aproveitamento dos recursos disponíveis (NEWMAN, 2015).

A divisão de responsabilidade tende a seguir diversas premissas, tendo como destaque o princípio da única responsabilidade, onde sugere manter juntas as unidades que sejam alteradas pela mesma razão, e separando as unidades que sejam alteradas por razões diferentes (MARTIN, 2003).

5. Refatoração

Refatoração é o processo que consiste em realizar uma mudança em um sistema de software de forma que não altere o comportamento externo do código, deixando com uma estrutura interna melhor. É uma forma de organizar o código e minimizar as chances de introdução de defeitos (Fowler, 1999), sendo capaz de ajudar a melhorar de maneira incremental os atributos de qualidade de um sistema (KATAOKA,2001).

A refatoração é organizada na forma de um catálogo de refatoração, que consiste em um nome, contexto aplicável, seu conjunto de etapas de aplicação e um ou mais exemplos de sua aplicação (FOWLER,1999; FOWLER, 2019).

Durante o processo de refatoração, códigos ou funções duplicadas, lógica condicional simplificada e segmentos de código alterados pouco claros serão identificados e excluídos. Essas alterações podem ser tão grandes quanto a unificação da hierarquia ou tão pequenas quanto alterar o nome da variável (KERIEVSKY, 2008).

Para garantir que o comportamento externo não tenha sido alterado ou que as falhas não tenham sido introduzidas, testes manuais ou automáticos são necessários. Para tanto, uma decomposição passo a passo ajuda a dividir o problema em partes menores e testá-las individualmente (KERIEVSKY, 2008).

Um dos motivos para refatorar o código-fonte é torná-lo mais provável de receber novos recursos. Ao adicionar um componente funcional, você pode programar sem se preocupar com a aplicabilidade do novo componente funcional ao código-fonte e também pode reestruturar a estrutura para acomodar o componente funcional. No primeiro caso, o aumento de funções trará obrigações para o projeto, e a dívida deve ser paga através da reconstrução (KERIEVSKY, 2008).

O processo de reestruturação constante do projeto tornará o processo de trabalhar com ele cada vez mais simples. Buscar problemas constantemente e resolvê-los imediatamente após descobri-los ajudará a melhorar a organização e a clareza do projeto. Portanto, o código-fonte será mais fácil de manter e estender. Além disso, o código pouco claro precisa ser refatorado. Dessa forma, outros desenvolvedores terão mais facilidade de acesso ao conhecimento expresso na função, facilitando a comunicação (KERIEVSKY, 2008).

5.1 Abordagens existentes para refatorar um Monólito para Microsserviços

Analisar o estado da arte é importante porque fornece insights sobre as tendências atuais e as soluções atuais para determinados problemas. Ao analisar o que outras pessoas descobriram para lidar com questões semelhantes, é mais fácil adotar uma estratégia ou implementar uma solução para um problema com características comparáveis. A realização desta análise também fornece uma análise de lacunas dos desafios atuais relacionados à migração de monólitos para microsserviços (PAIVA, 2019).

5.1.1 Estratégias e abordagens de decomposição

Atualmente, o processo de refatoração de uma arquitetura monolítica em uma arquitetura de microsserviços é um processo executado manualmente na maioria das vezes. Normalmente acontece porque a solução que está sendo refatorada é geralmente

uma aplicação legado que está muito entrelaçada e mal construída precisando de uma abordagem personalizada e específica. Outras vezes sendo porque as pessoas que fazem a refatoração não estão cientes de que existe uma alternativa para o processo manual. Algumas alternativas foram desenvolvidas ao longo do tempo que seguem estratégias diferentes e têm resultados diferentes. Alguns deles se concentram em formas de extrair serviços de monólitos (Baresi, Garriga, Renzis, 2017). Outros se concentram em maneiras de separar o monólito e identificar seus pontos de quebra (Mazloumi, Cito, Leitner, 2017).

A maior parte das abordagens de decomposição formal existentes podem ser agrupadas nas seguintes classificações (Fritzsch, Bogner, Zimmermann, Wagner, 2018):

- Abordagens que usam ferramentas de análise de código estático, exigindo, portanto, o código-fonte da aplicação. A partir dessa análise, as decomposições são geradas, podendo ter estágios intermediários necessários.
- Abordagens que usam metadados disponíveis e requerem, por exemplo, dados de entrada mais abstratos, como diagramas, interfaces ou histórico de controle de versão.
- Abordagens focadas em dados de carga de trabalho que tentam encontrar decomposições de serviço adequadas medindo o uso de dados operacionais, como desempenho ou métricas de comunicação, e usam esses dados para determinar uma decomposição e granularidade de serviço apropriadas.
- Abordagens de composição dinâmica de microsserviços tentam resolver o problema de decomposição em um ambiente de execução, cujo um conjunto de serviços resultantes muda permanentemente ao final de cada iteração através de um recálculo de composições apropriadas que podem ser baseadas, por exemplo na carga de trabalho.

As estratégias acima mencionadas são utilizadas para identificar pontos de ruptura do monólito de forma mais formal, permitindo que ferramentas possam sugerir pontos de ruptura que levariam a interfaces de serviço e possíveis candidatos a microsserviços (Fritzsch, Bogner, Zimmermann, Wagner, 2018).

Além dessas abordagens mais formais encontradas na literatura, as abordagens de decomposição mais comuns usadas na indústria são baseadas em capacidades de negócios (Richardson, 2018) e contextos limitados (de Domain Driven Design).

O primeiro usa uma estratégia para transformar um monólito em um conjunto de microsserviços, decompondo-os por capacidade de negócios. Em essência, uma capacidade de negócios pode ser definida como algo que é realizado no contexto de um negócio que gera valor e geralmente captura o que é o negócio de uma organização (Richardson, 2018).

Uma capacidade de negócio pode ser identificada através da análise do propósito, estrutura e processos de negócios da organização. Depois que os recursos de negócios são identificados, eles são frequentemente promovidos para serviços por um mapeamento um a um ou por um acordo de serviço com um grupo de recursos de negócios. Devido à relativa estabilidade dos recursos de negócios, a criação de serviços

a partir deles tem a vantagem de que os serviços e sua respectiva arquitetura também devem ser estáveis (Richardson, 2018).

Outra estratégia que pode ajudar a criar serviços a partir de monólitos existentes tem a ver com o conceito de subdomínios e contextos limitados, que são dois conceitos incrivelmente úteis que vêm do DDD. Em vez de usar uma abordagem tradicional em que há um único modelo de domínio para todo o sistema, o DDD usa modelos de domínio separados com um escopo menor para cada subdomínio identificado, o que é útil porque significa que uma organização inteira não precisa concordar com a definição de apenas um modelo. Isso evita o uso de modelos de domínio excessivamente complexos em situações em que uma representação muito mais simples seria suficiente, além de fornecer flexibilidade no sentido de que diferentes subdomínios podem se referir a diferentes conceitos ou termos com o mesmo nome e isso elimina o problema. Os subdomínios são identificados de forma semelhante às capacidades de negócios mencionadas anteriormente: por meio da análise do negócio e da identificação das diferentes áreas de especialização. O escopo de um modelo de domínio é chamado de contexto abundante. Um contexto limitado é um candidato ideal para se tornar um microsserviço devido ao seu escopo limitado e representação focada no domínio (EVANS, 2014).

5.2 Padrão de migração

Os padrões são geralmente definidos como soluções para problemas comuns e recorrentes. Eles são úteis para desenvolvedores, arquitetos ou qualquer pessoa que esteja envolvida de alguma forma no desenvolvimento de softwares porque podem ser usados como uma solução comprovada para um determinado problema e facilitar o desenvolvimento (Salingaros, 2000). Os padrões de migração têm o mesmo princípio fundamental, mas são mais voltados para problemas relacionados à migração de sistemas de um para outro, como é o caso da refatoração de monólitos em microsserviços.

Transformar uma arquitetura monólito em uma arquitetura nativa da nuvem, como microsserviços, envolve uma série de etapas que são frequentemente repetidas sempre que há a necessidade de migrar tal sistema. As seções a seguir descrevem algumas estratégias de migração comuns encontradas na literatura que derivam de cenários e requisitos da vida real e que são comuns à maioria das soluções de microsserviços.

5.2.1 Integração Contínua

A necessidade de ter pipelines automáticos de construção e teste que resultam em artefatos de entrega aumenta à medida que a complexidade de um sistema aumenta. Em um monólito, não ter essas tarefas automatizadas é algo gerenciável, mas em uma arquitetura de microsserviços, a falta de integração contínua torna muito difícil e complexo gerenciar compilações e testes e ter confiança na qualidade do código que está sendo produzido (Mouat, 2015). Habilitar a integração contínua não só torna isso mais gerenciável, mas também permite adicionar implantação contínua em uma etapa futura (Balalaie, Heydarnoori, 2018).

5.2.2 Implantação Contínua

Uma das maiores certezas no desenvolvimento de software é a entrega de novos recursos ou correções de bugs, no que geralmente pode ser referido como atualizações. Automatizar o processo de entrega é essencial para o ciclo de vida do software e nos dá mais confiança de que as alterações que queremos enviar serão enviadas corretamente para onde queremos (Mouat, 2015). Executar manualmente a liberação de atualizações está sujeito a erros e deve ser automatizado. Isso é ainda mais evidente no caso de microsserviços, em que cada microsserviço individual deve ter um pipeline de implantação contínua separado para facilitar o processo de liberação e atualização (Susan, 2016).

5.2.3 Decompondo o Monólito

Conforme mencionado anteriormente, ao falar sobre estratégias de decomposição, decompor o monólito é uma das etapas mais significativas ao realizar a migração. Existem várias estratégias e técnicas que podem ser levadas em consideração, mas o resultado final será sempre a divisão do monólito em um conjunto de serviços diferentes com a menor entropia possível (Susan, 2016).

5.2.4 Alterar a dependência do código para chamada de serviço

Durante o processo de migração e divisão do monólito, haverá situações em que uma chamada de código deve ser alterada para uma chamada de serviço. Isso normalmente acontece quando a lógica que suporta a chamada do método que agora está em um microsserviço diferente. Para preservar a funcionalidade, a chamada do método deve ser alterada para uma chamada de procedimento remoto (remote procedure call, RPC). Normalmente, existem dois tipos de visão diferentes entre como implementar um RPC. Um estilo síncrono é usado com protocolos como HTTP e gRPC ou um estilo assíncrono com mensagens com tecnologias, por exemplo, como RabbitMQ ou Apache Kafka (Susan, 2016).

5.2.5 Service Discovery

Em um ambiente de nuvem, geralmente há um alto grau de dinamismo em relação aos serviços expostos. Os serviços não são estáticos. Eles podem ser replicados ou desligados devido à elasticidade que este tipo de arquitetura oferece. O Service Discovery permite que os serviços se registrem à medida que são iniciados em um registro estático que é então consumido pelas partes interessadas. A remoção de um serviço do registro pode ser acionada pelo serviço que não responde a uma verificação de pulsação periódica ou pelo próprio serviço (Susan, 2016).

5.2.6 Balanceamento de carga

O uso de balanceamento de carga é frequentemente usado para dimensionar aplicativos monolíticos replicando o monolito e, em seguida, usando um balanceador de carga para decidir qual monolito deve ser usado. Em um sistema distribuído como os microsserviços, essa necessidade é ainda mais evidente devido à possibilidade de haver replicação na maioria dos serviços, o que é uma consequência direta desse tipo de arquitetura elástica nativa da nuvem. Por isso, a necessidade de gerenciar todas as

réplicas e decidir qual usar é ainda maior neste cenário, o que torna quase obrigatório o uso de um balanceador de carga (Susan, 2016).

5.2.7 Edge Server

Um Edge Server é o ponto de entrada de um sistema distribuído. Ele atua como uma fachada em todos os microsserviços existentes e também é geralmente usado para rotear o tráfego, atuar como um proxy ou proxy reverso. No contexto de microsserviços, isso geralmente é chamado de gateway de API. A importância de ter um Edge Server torna-se evidente em uma arquitetura de microsserviços devido ao dinamismo dos serviços. Os clientes não devem ter que conhecer todos esses serviços. Eles só devem estar cientes de um ponto de entrada no sistema de onde podem operar e usá-lo.

5.2.8 Dividindo o banco de dados

Um dos maiores desafios ao realizar a mudança de monólitos para microsserviços é decidir o que deve ser feito em relação ao banco de dados. Normalmente, os sistemas monolíticos usam uma estratégia de banco de dados compartilhado em todo o aplicativo, o que significa que há um forte acoplamento entre o aplicativo inteiro e o banco de dados. Uma arquitetura típica em aplicativos monolíticos é ter uma camada de repositório que expõe métodos que tornam possível realizar ações controladas sobre o banco de dados e então usar um ORM como Hibernate ou Entity Framework, para mapear Entidades OO (Orientado a Objetos) para suas respectivas representações no banco de dados, geralmente uma tabela (NEWMAN, 2019).

Ao transformar um sistema monolítico em microsserviços, deve-se tomar uma decisão: ou o banco de dados é compartilhado por todos os microsserviços ou deve ser dividido e transformado em diferentes bancos de dados, cada um usado por cada microsserviço (NEWMAN, 2019).

Optar pela primeira abordagem tem algumas vantagens (NEWMAN, 2019):

- Todo o banco de dados pode ser acessado com consultas SQL, o que significa que não há problema em acessar informações muito distantes e que estariam em outro microsserviço se a opção de usar um banco de dados por microsserviço fosse adotada.
- Todas as transações realizadas no banco de dados são ACID, o que torna muito mais fácil lidar com as transações.
- Um único banco de dados é mais simples de operar.

No entanto, pode não ser a melhor abordagem em uma arquitetura de microsserviços devido às seguintes desvantagens (NEWMAN, 2019).:

- Um desenvolvedor trabalhando em um serviço precisará coordenar as mudanças de esquema com desenvolvedores de outros serviços que tenham acesso às mesmas tabelas. Este acoplamento e coordenação adicional irão desacelerar o desenvolvimento.
- Como todos os serviços acessam o mesmo banco de dados, eles podem interferir uns nos outros. Por exemplo, se uma transação de longa duração mantém um bloqueio em uma tabela, o microsserviço que usa essa tabela pode ser potencialmente bloqueado.

- Um único banco de dados pode não satisfazer os requisitos de armazenamento de dados e acesso de todos os serviços.

A outra abordagem é dividir o banco de dados em bancos de dados de microsserviços específicos. As abordagens mais comuns são usar uma tabela independente, esquema independente ou servidor de banco de dados independente por microsserviço. Usar um banco de dados por microsserviço vai de encontro à filosofia de desenvolvimento de microsserviços (NEWMAN, 2019). Tudo está contido em um microsserviço, com cada microsserviço tendo total propriedade e apenas conhecendo seu banco de dados.

As vantagens mais comuns desta situação são:

- Cada microsserviço pode escolher a solução de persistência mais apropriada para seus negócios específicos ou necessidades de domínio (por exemplo, usando um ElasticSearch para fornecer funcionalidade de pesquisa de texto completo avançada ou usando Neo4j para um caso de uso intensivo de gráficos).
- Reforçando o acoplamento fraco entre diferentes microsserviços.

Tendo bancos de dados distribuídos, embora vantajosos, também possuem problemas que decorrem da complexidade introduzida por esta abordagem, mais especificamente:

- Implementar transações de negócios que abrangem microsserviços diferentes torna-se mais desafiador porque não pode ser feito recorrendo a uma única transação ACID. Em vez disso, diferentes mecanismos devem ser levados em consideração, como transações distribuídas, orquestração ou composição de APIs ou uma arquitetura publish-subscribe com mensagens de eventos.
- A complexidade de gerenciar bancos de dados diferentes aumenta devido às particularidades de como cada banco de dados funciona (por exemplo, NoSQL vs bancos de dados relacionais)

Do ponto de vista da transformação de refatoração, escolher a primeira escolha é obviamente mais fácil, não requer transformação às custas das desvantagens mencionadas. Em contraste, escolher o banco de dados por abordagem de serviço, embora complicado, pode ser muito mais vantajoso em uma longa execução (NEWMAN, 2019).

6. Projeto PAR

Em um ambiente de trabalho é comum existirem algumas atividades que saem dos limites de apenas um cargo, tornando-se uma atividade multifacetada. Essas atividades podem ser feitas por mais de uma pessoa sendo elas de diferentes cargos. Essas atividades são contabilizadas e adicionadas em um plano que define a quantidade horas de trabalho um recurso pode realizá-la, em um período semanal e depois e registrar durante um período semestral.

O PAR (Plano de atividades do recurso) é um modelo de sistema que coleta as informações de um recurso de um ambiente de trabalho, definindo quais atividades o seu recurso irá executar em um período de tempo. Essa plataforma foi desenvolvida para analisar e determinar quais atividades um recurso deve executar, e qual foi seu nível de excelência aplicando e atividade escolhida ou designada a ele.

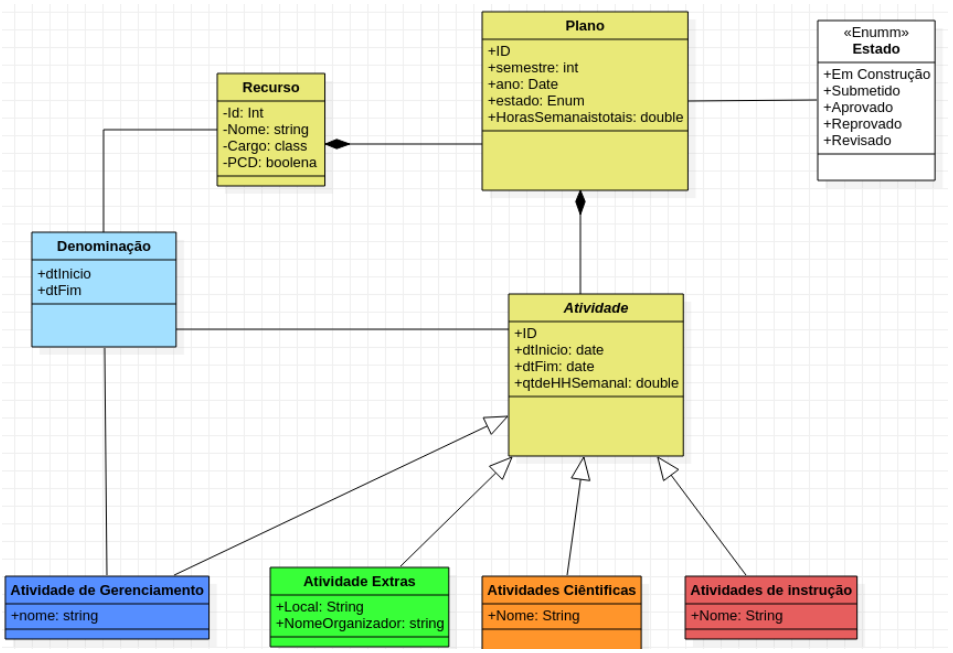

Figura 2. Diagrama de classe do PAR

Fonte: Elaborado Pelos Autores

6.1 Arquitetura do Projeto

Seguindo o padrão de desenvolvimentos de aplicações web sendo elas projetadas para não sofrerem alteração em um período longo de desenvolvimento, o sistema PAR foi construído sobre a estrutura na qual:

- Todos os serviços estavam contidos em um núcleo indecomponível que permitia a comunicação irrestrita entre os componentes.
- Todo o sistema era executado em um único nó de processamento.
- Havia um acúmulo de múltiplas responsabilidades como a renderização da camada de apresentação, controle de estados do objeto complexos, longos processamentos (síncronos) em background, processamento de lógica de negócio, acesso ao banco de dados e outros componentes de infraestrutura, ilustrado na figura 3.

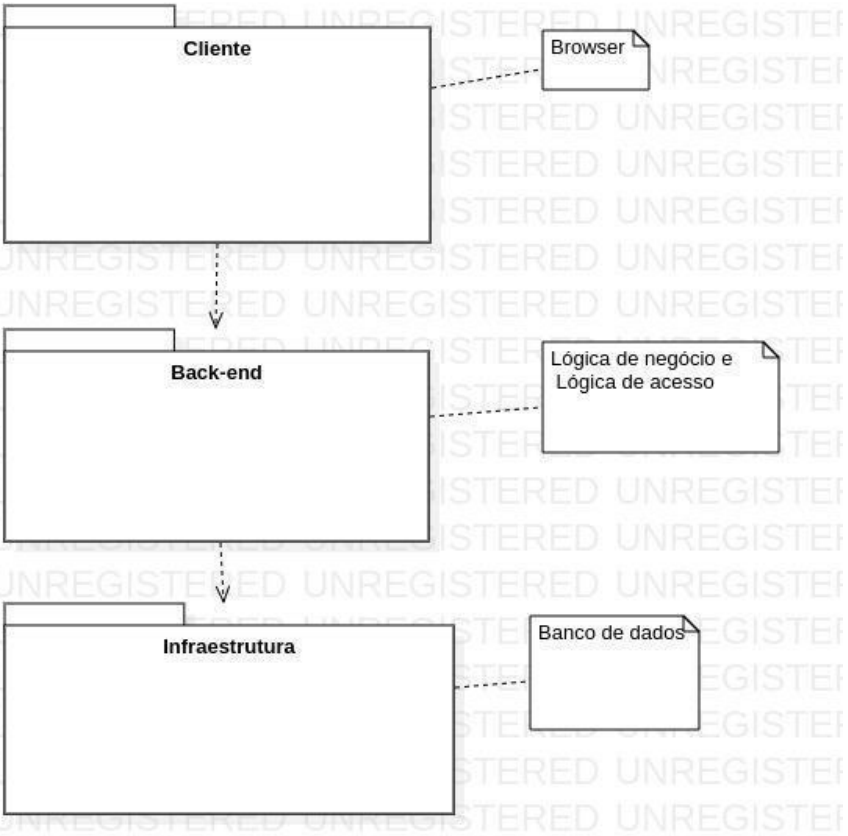

Figura 3. Arquitetura monolítica do PAR

Fonte: Elaborado Pelos Autores

7. Desenvolvimento

Um dos primeiros aspectos que é necessário ter de considerar como parte da migração é se realmente é possível realizar a mudança do sistema monolítico atual. Em algumas situações podem haver algumas restrições mais rígidas, e não será possível ter essa oportunidade como foi apresentado nos tópicos anteriores. A maior barreira para fazer o uso do código existente no sistema monolítico nos microsserviços é o fato de as bases de código não estarem tradicionalmente organizadas em torno dos conceitos de domínio de negócio (NEWMAN, 2019).

7.1 Padrão Strangler Fig

Após o processo de mapeamento do sistema que foi apresentado nas figuras 2 e 3, inicia-se o processo migração do sistema. Newman aponta que existem muitas técnicas usadas como parte de uma migração para microsserviços. A técnica strangler fig é vista com muita frequência na reescrita de sistemas (NEWMAN, 2019). O sistema consiste em remover pequenas partes do sistema, fazendo com que o velho consuma o novo gradativamente, até que o novo sistema tenha inicialmente suporte do sistema existente e o encapsule, fazendo com que o sistema velho e o novo possam coexistir, dando tempo ao novo sistema para que cresça e, possivelmente, substitua totalmente o sistema antigo (NEWMAN, 2019).

A principal vantagem desse padrão é que ele está alinhado com o objetivo de permitir uma migração gradual para um sistema novo. O padrão permite fazer pausas, e até mesmo interromper totalmente a migração, enquanto os recursos possam utilizar as vantagens do novo sistema disponibilizado até então (NEWMAN, 2019).

O padrão strangler fig embora ele seja utilizado para migração de um sistema monolítico para outro, o objetivo deste projeto é fazer uma migração de um sistema monolítico para uma série de microsserviços. Implementar um padrão strangler fig depende de três passos. O primeiro passo é realizar a identificação das partes do sistema atual cuja migração será feita.

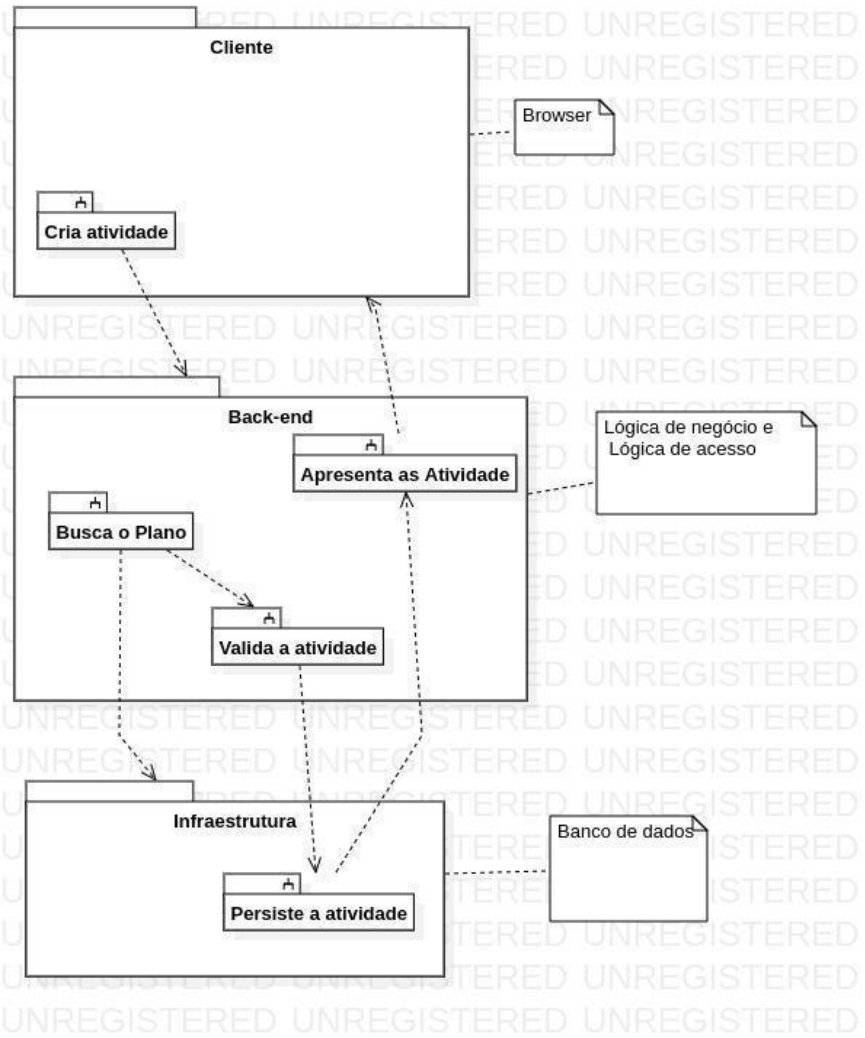

Figura 4. Processo de criação de uma atividade do recurso

Fonte: Elaborado Pelos Autores

Na figura 4, podemos ver o processo de criação de uma atividade, onde o recurso adiciona as informações da atividade e submete essa atividade ao seu plano atual. Após a submissão da atividade, o back-end realiza o processo de identificar o plano do recurso em que a atividade foi inserida, e após encontrar o plano é feita a análise e a persistência da atividade no banco de dados.

Observe que o plano somente é consultado para realizar uma busca de qual plano está sendo inserida a nova funcionalidade, tornando se o serviço mais viável de ser separado do sistema.

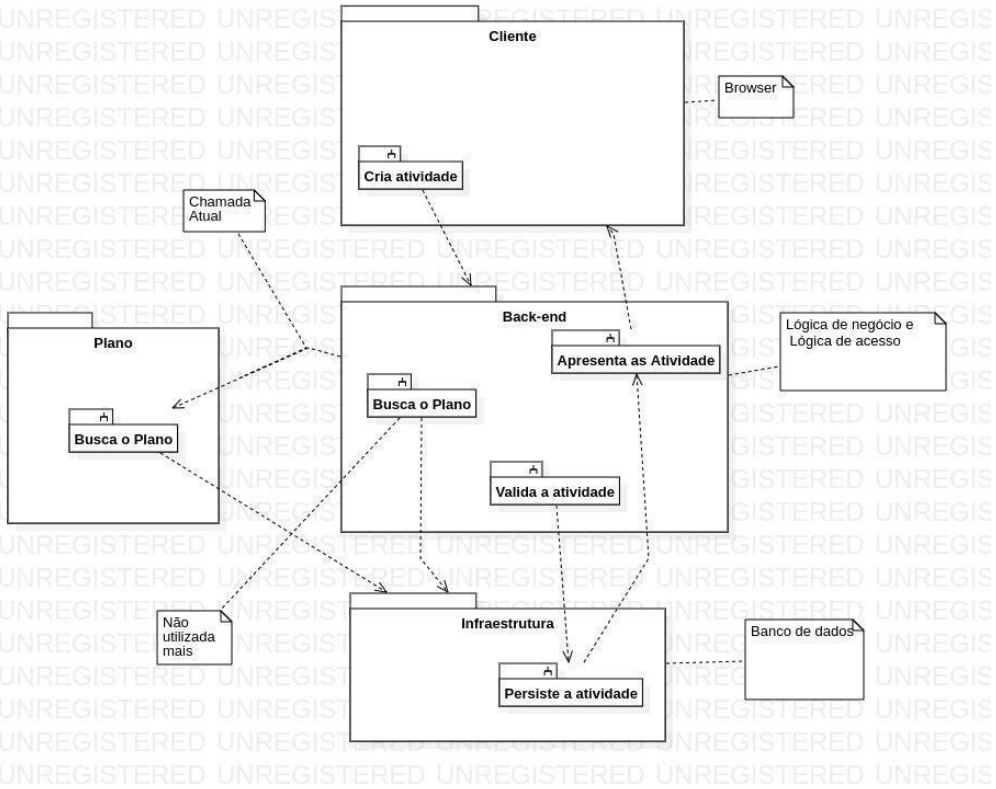

Figura 5. Arquitetura do PAR com o microsserviço do plano

Fonte: Elaborado Pelos Autores

Observe que até que a chamada para a função movida seja redirecionada, mesmo se a nova função foi implementada no ambiente de produção, a nova função não estará tecnicamente ativa. Isso significa que, com o tempo, o sistema pode funcionar normalmente durante a implementação antes de funcionar corretamente.

Para realizar o chamado processo de atividade entre o todo e o serviço planejado, é necessário estabelecer um modo de comunicação entre eles. Protocolos como HTTP são muito adequados para redirecionamento. O próprio HTTP tem o conceito de redirecionamento transparente e um proxy pode ser usado para entender claramente a natureza da solicitação de entrada e contorná-la de maneira adequada.

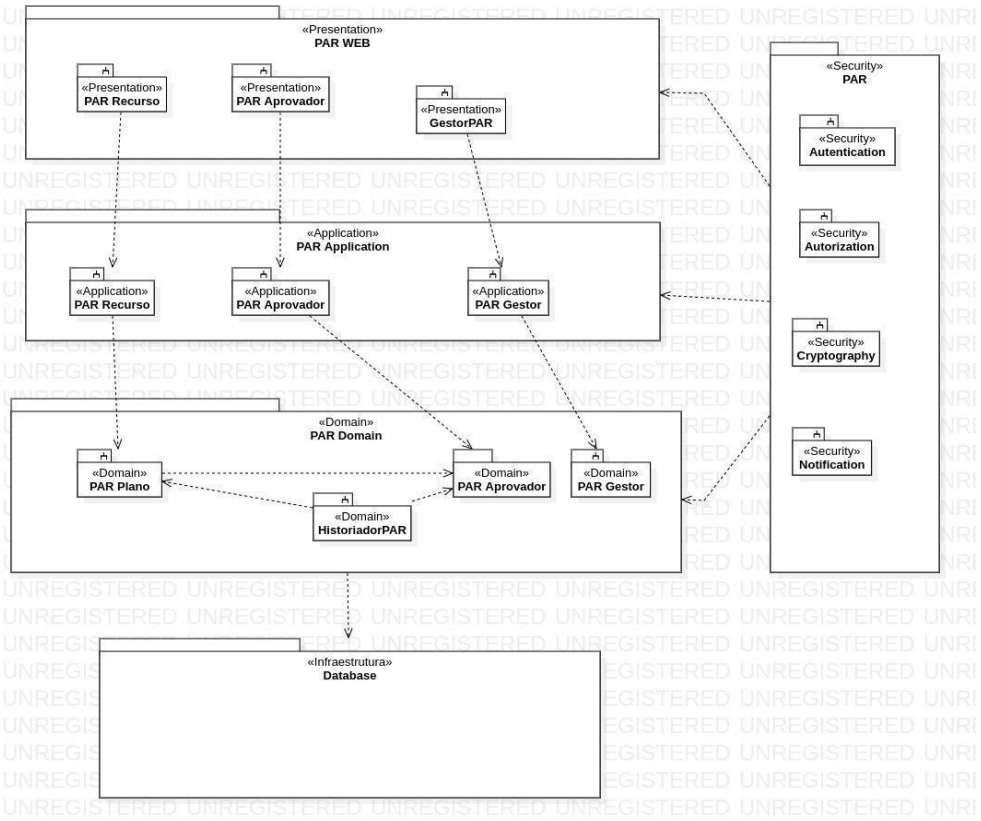

Figura 6. Arquitetura do PAR em microsserviços

Fonte: Elaborado Pelos Autores

A Figura 6 apresenta a migração completa dos serviços do monolito para uma arquitetura que adota os microsserviços apresentados por camadas, sendo cada componente responsável por reger uma tarefa designada a ele.

7.2 Decompondo o Banco de dados

A Figura 6 mostra a migração completa de um serviço monolítico para uma arquitetura que usa provisão hierárquica de Microsserviços. Cada componente é responsável por gerenciar as tarefas atribuídas.

Conforme mostrado na Figura 6, a migração de serviços foi concluída e cada componente tem sua própria tarefa. No entanto, observe que todos os serviços se referem ao mesmo banco de dados inicial do banco de dados geral. Isso nos leva ao próximo estágio da migração, que é a decomposição do banco de dados. Como apresentado nos tópicos anteriores, a separação do banco de dados neste caso, tende a ser uma separação lógica. Um único mecanismo de banco de dados é totalmente capaz de hospedar várias arquiteturas separadas logicamente.

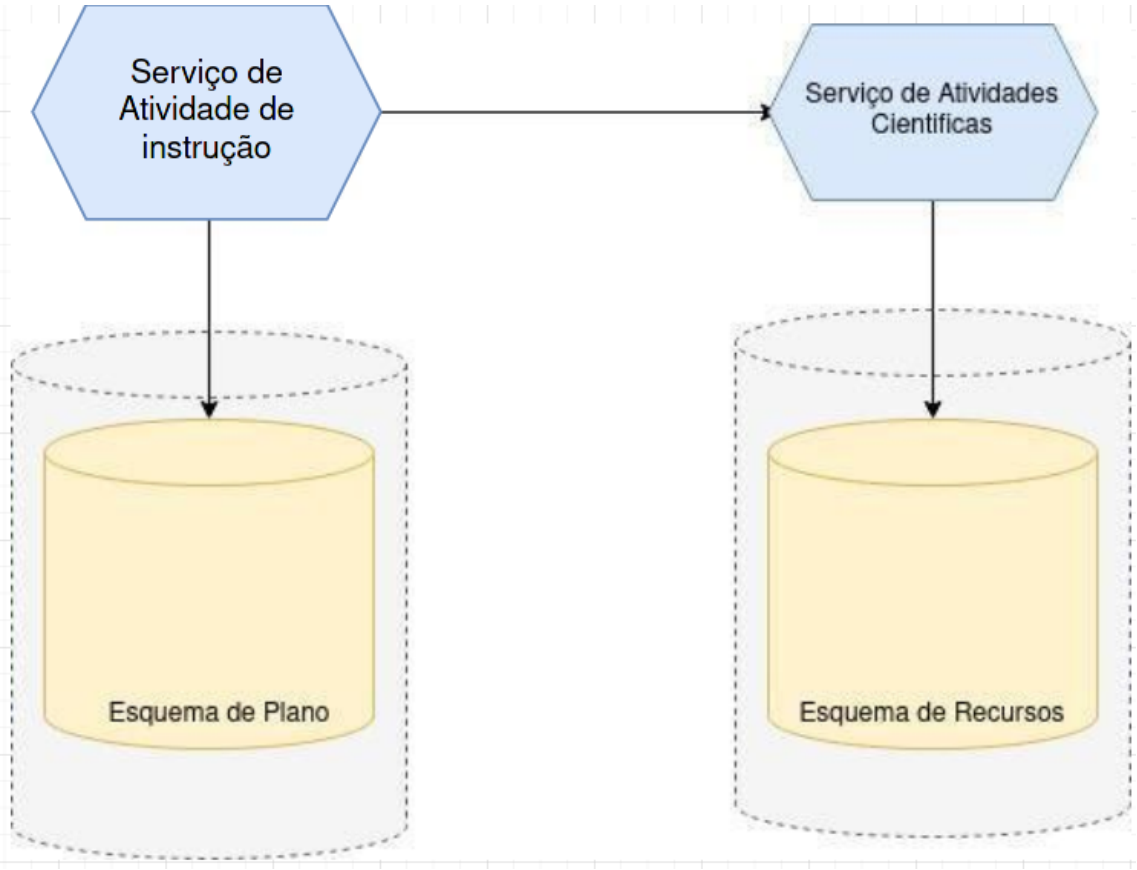

Figura 7. Dois serviços fazendo uso de esquemas separados do ponto de vista lógico, ambos executados sua própria engine física de banco de dados.

Fonte: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith

Uma decomposição lógica permite mudanças independentes de modo mais simples e possibilita ocultar informações, enquanto uma decomposição física pode melhorar a

robustez do sistema e poderia ajudar a acabar com a contenção de recursos para melhorar o throughput ou diminuir a latência.

Para realizar o processo de decomposição do banco de dados foi adotado o padrão banco de dados por contexto delimitado. Esse padrão apresenta uma maneira de decompor o banco por delimitações, onde o contexto delimitado tem seu próprio banco de dados totalmente separado. E caso haja a necessidade de uma separação de microsserviços mais tarde, essa seria uma tarefa muito mais fácil.

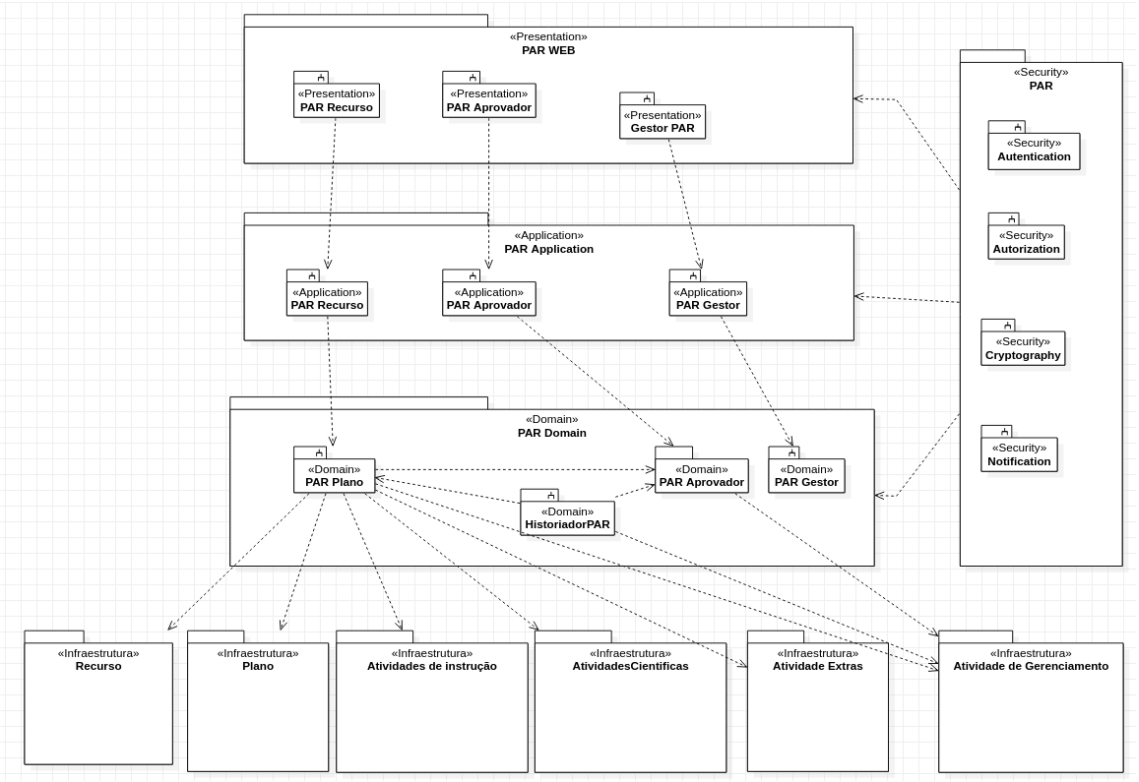

Figura 8. Decomposição do banco de dados aplicado na camada de infraestrutura

Fonte: Elaborado Pelos Autores

8. Considerações Finais

Este projeto foi desenvolvido com o objetivo de auxiliar engenheiros e desenvolvedores que atualmente aplicam o processo de migração de monólitos para microsserviços. Através de uma revisão da literatura existente e uma análise do estado da arte atual em relação a monólitos para microsserviços apresentando um sistema de planejamento de recursos genérico para lidar com a migração de um modelo projeto que adota o padrão de arquitetura monolítica e alcançar uma solução com o padrão arquitetural de microsserviços. Essa implementação proporciona um modelo de arquitetura que garante a liberdade da equipe de desenvolvimento na escolha de tecnologia, garante uma facilidade em resolver problemas de manutenção, escalabilidade e disponibilidade. Com relação a possíveis melhorias e limitações e suposições atuais, o trabalho futuro

relacionado aos detalhes de implementação da ferramenta de refatoração pode ser resumido nos itens a seguir:

- Como realizar o processo de Deploy de uma aplicação que adota o padrão de microsserviços.
- Como aplicar tecnologias de diferentes tipos em um mesmo projeto usando o padrão de arquitetura de microsserviços.

Referências

Adrian Mouat.Using Docker: Developing and Deploying Software with Containers. "O'Reilly Media, Inc.", 2015

Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, and Theo Lynn.Microservices migration patterns.Software: Practice and Experience, 2018.

Chris Richardson. Pattern:Monolithic architecture. Disponível em: https://microservices.io/patterns/monolithic.html. Acesso em 30-05-2021.

D. ALINE, P. DANIEL, ANTONIO J. Design Science Research: Método de Pesquisa para Avanço da Ciência e Tecnologia. January 2015.

DAYA, SHAHIR, NGUYEN VAN DUY, KAMESWARA EATI, CARLOS M. FERREIRA, DEJAN GLOZIC, VASFI GUCER, MANAV GUPTA, SUNIL JOSHI, VALERIE LAMPKIN, MARCELO MARTINS. Microservices from 120 Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach. 1. ed. [s.l.] IBM Redbooks, 2016

DI FRANCESCO, P. Architecting microservices . In 1st International Conferenceon Software Architecture Workshops (ICSAW), pages 224–229, 2017.

Eric Evans.Domain-Driven Design Reference: Definitions and Pattern Summaries. Dog EarPublishing, 2014.

F. Bucshmann, K. Henney, and D. C. Schmidt, Pattern-oriented software architecture. Chichester: Wiley, 1996.

FOWLER, M. AND JAMES LEWIS. Microservices, a definition of this new architectural term, 2014. Disponível em: https://martinfowler.com/articles/microservices.html.

FOWLER, M. Microservice Premium,2015. Disponível em: https://martinfowler.com/bliki/MicroservicePremium.html.

FOWLER, M. Microservice Prerequisites,2015. Disponível em: https://www.martinfowler.com/bliki/MicroservicePrerequisites.html.

FOWLER, M. Microsservice Trade-Offs,2015. Disponível em: https://www.martinfowler.com/microservices/

FOWLER, M. Monolith First,2015. Disponível em: https://www.martinfowler.com/microservices/

FOWLER, Martin. Patterns of Enterprise Architecture. Addision-Wesley, 2002.

John Jeston and Johan Nelis.Business Process Management: Practical Guidelines to Suc-cessful Implementations. 2006.

Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: a classification of refactoring approaches. InInternational Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, pages 128–141. Springer, 2018.

KATAOKA, Y. et al. Automated Support for Program Refactoring Using Invariants. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'01).Procee-dings. . .[S.l.: s.n.], 2001. p.736.

KERIEVSKY, J.Refatoração para Padrões. Porto Alegre: Bookman, 2008.

KRAFZIG, D. ,BANKE, K. & SLAMA, D. Enterprise SOA: Service-Oriented Architecture Best Practices. Indianapolis: Prentice Hall, 2004.

Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices Identification ThroughInterface Analysis. pages 19–33. Springer, Cham, 2017

MACHADO, M. G. Micro Serviços: Qual a diferença para arquitetura monolítica?, 2017. Disponível em: https://www.opus-software.com.br/micro-servicos-arquietura-monolitica/

MARTINS, J. C. C. Técnicas para Gerenciamento de Projetos de Software, Brasport, cap. A, p. 18, 2007.

M. RICHARDS. Microservices vs Service Oriented Architecture. O'Reilly Media, Inc., 2015

NAMIOT, D. On micro-services architecture ,2014. Disponível em: https://www.researchgate.net/publication/268919364_Micro-service_Architecture_for_Emerging_Telecom_Applications.

NAMIOT, D. E MAFRED SNEPS-SNEPPE. Micro-service Architecture for Emerging Telecom Applications, 2014. Disponível em: https://www.researchgate.net/publication/268919364_Micro-service_Architecture_for_Emerging_Telecom_Applications.

Nikos A. Salingaros. The structure of pattern languages.Architectural Research Quarterly,2000.

K. PEFFERS, T. TUUNANEN, M. ROTHENBERGER, S. CHATTERJEE. A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems, Vol.24, No.3 2007.

RICHARDSON, C. Pattern: Microservices Architecture, 2019. Disponível em: https://microservices.io/patterns/microservices.html

RICHARDSON, C. Microservices: Decomposing Applications for Deployability and Scalability,2014. Disponível em: https://www.infoq.com/articles/microservices-intro/.

RICHARDSON, C. Pattern: Monolithic Architecture, 2019. Disponível em: https://microservices.io/patterns/monolithic.html

ROUSE, M. Monolithic Architecture, 2016. Disponível em: https://whatis.techtarget.com/definition/monolithic-architecture

SORDI, J.O., MARINHO, B.L. & NAGY, M. Benefícios da Arquitetura de Software Orientada Serviços para Empresas: Análise da Experiência do ABN AMRO Brasil. Revista de Gestão da Tecnologia e Sistemas de Informação, 2006.S.

S. NEWMAN. “Building Microservices”. O’Reilly Media, Inc., 2015.

S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, 1, O'Reilly Media, Inc., Estados Unidos, 2019.

SOMMERVILLE, I.Engenharia de Software, Tradução de André Maurício de Andrade Ribeiro; Revisão técnica de Kechi Hirama. [S.l São Paulo, Addison Wesley, 2003.

SOMMERVILLE, I. Engenharia de software, 8a edição, tradução: Selma shin shimizu melnikoff, reginaldo arakaki, edilson de andrade barbosa.São Paulo: Pearson Addison-Wesley,v. 22, 2007.

SOMMERVILLE, I.Software Engineering. [S.l Addison Wesley, São Paulo, 9th edition, 2009.

Susan J Fowler.Production-ready microservices: Building standardized systems across an engineering organization. " O’Reilly Media, Inc.", 2016.

PRESSMAN, R. S. Engenharia de software: uma abordagem profissional. 7a edição. Ed:McGraw Hill, 2011.

PRESSMAN, S. R.Engenharia de Software. [S.l.]: McGraw Hill, 6th edition, 2005.

The Open Group. SOA Source Book. Disponível em http://www.opengroup.org/soa/source-book/intro/index.htm, 2011. Acessado em 30-05-2021.

Viggiato, M.; Terra, R.; Valente, M.; Figueiredo, E., “Microservices in Practice: A Survey Study”, 2018.