TRABALHO

MICROSERVICES & SERVERLESS ARCHITECTURE

SÃO PAULO - SP

2021

Trabalho Microservices & Serverless Architecture

ALBERTO SOUSA DE SANTANA

ITALO FERNANDES

THIAGO MARINHO DA SILVA

WESLEY DA SILVA VASCONCELOS

TRABALHO TEÓRICO SOBRE PRINCIPAIS DIFERENÇAS ENTRE OS ESTILOS ARQUITETURAIS DE MICROSERVICES, SOA E MONOLÍTICO

SÃO PAULO - SP

2021

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

NENHUMA ENTRADA DE ÍNDICE DE ILUSTRAÇÕES FOI ENCONTRADA.

RESUMO

Atualmente, as empresas almejam que os processos de negócios estejam cada vez mais alinhados com Tecnologia da Informação – (TI). Um número considerável de soluções baseadas em software foram apresentadas nos últimos anos a fim de atingir este objetivo. Umas destas soluções é a arquitetura orientada a serviços - (SOA) que possui uma proposta que promove uma significativa melhora na forma com que os processos de negócio são transformados em funcionalidades de software. SOA é um paradigma de desenvolvimento de sistemas distribuídos que possibilita independência de tecnologia e promove o reuso de processos de negócio na forma de serviços.

Os microsserviços surgiram para preencher algumas lacunas deixadas pela arquitetura orientada a serviços (SOA) com soluções baseadas em nuvem, promovendo uma sensível melhora no cumprimento de requisitos não funcionais, principalmente relacionados a desempenho, escalabilidade e disponibilidade.

Palavras-chave: SOA. BPM. Processos. Arquitetura orientada a serviços. Microsserviços.

ABSTRACT

Currently, companies crave that business processes are increasingly aligned with Information Technology - (IT). A considerable number of solutions have been put forward in recent years in order to achieve this goal. Of these solutions is the service-oriented architecture - (SOA) that has a proposal that promotes a significant improvement in the way business processes are transformed into software features (business processes). SOA is a distributed systems development paradigm that enables independent technology and promotes the reuse of business processes as services. Microservices emerged to fill some gaps left by the service-oriented architecture (SOA) with cloud-based solutions, promoting a significant improvement in the fulfillment of non-functional requirements, mainly related to performance, scalability and availability.

Keywords: SOA. BPM. Process. Service Oriented Arquitetury. Microservices.

1. INTRODUÇÃO

Em um mercado extremamente competitivo, as organizações procuram apresentar soluções rápidas frente às necessidades dos clientes. O setor de Tecnologia da Informação – (TI) vem colaborando para que as empresas possam atingir este objetivo, desenvolvendo um conjunto considerável de soluções baseadas em software com intuito de atender os mais diversos processos de negócios.

No mundo dos negócios, faz muito sentido entregar soluções capazes de automatizar a execução de tarefas de negócio (Erl, 2009).

Uma organização de médio e grande porte possui diversos departamentos ou setores, os quais em geral utilizam diferentes aplicações para realizar suas atividades. Estas aplicações necessitam se comunicar de forma integrada com o objetivo de atingir agilidade e simplificar processos de negócio, tornando-os mais produtivos, frente à crescente e a intensa competitividade do mercado.

A arquitetura orientada a serviços – (SOA) é um paradigma de desenvolvimento de sistemas distribuídos que permite integrar aplicações independentemente de linguagens de programação, plataformas ou sistemas operacionais, onde os serviços computacionais chamados de *web services* são construídos aplicando protocolos-padrão amplamente utilizados na internet. SOA possui as melhores práticas para integração de aplicações, fazendo uso de conceitos que foram utilizados com sucesso em implementações baseadas em componentes e integrações de aplicativos coorporativos. (Keen, Bond, *et al.*, 2005).

O objetivo de SOA é permitir às organizações automatizarem seus negócios por intermédio de serviços reutilizáveis buscando um melhor alinhamento entre TI e negócios. (Bieberstein, 2008). Um dos benefícios mais importantes que SOA fornece é a melhora da produtividade e agilidade de resposta ao negócio, contribuindo diretamente para a redução de custos no desenvolvimento e manutenção dos sistemas envolvidos. Sommerville resume o potencial das abordagens orientadas a serviços Sommerville (2011, p.358 apud Newcomer e Lomow, 2005) como:

Impulsionada pela convergência de tecnologias-chave e a adoção universal de web services, a empresa orientada a serviços promete melhorar significativamente a agilidade empresarial, a velocidade da inserção de novos produtos e serviços no mercado, reduzir custos e melhorar a eficiência operacional.

Com início da computação em nuvem, surgiram nova soluções computacionais para automatização de processos de negócio que proporcionaram um melhor aproveitando do poder computacional da infraestrutura das empresas.

A arquitetura de microsserviços tornou-se referência no modelo de computação distribuída. Os microsserviços surgiram para preencher algumas lacunas deixadas pela arquitetura orientada a serviços (SOA) com soluções baseadas em nuvem, promovendo uma sensível melhora no cumprimento de requisitos não funcionais, principalmente relacionados a desempenho, escalabilidade e disponibilidade.

Levando em consideração a história da SOA, os microsserviços não são uma ideia completamente nova. Porém, eles se tornaram mais viáveis graças aos avanços nas tecnologias de conteinerização. Com os containers Linux, agora é possível executar várias partes de uma aplicação de maneira independente no mesmo hardware e com um controle muito maior sobre os componentes individuais e ciclos de vida. Juntamente com as APIs e as equipes de DevOps, os microsserviços em containers são os pilares das aplicações nativas em cloud.

Sam, Newman define a arquitetura de microsserviços como:

A arquitetura de microsserviços promove a criação de serviços granulados com seus próprios ciclos de Vida, que colaboram juntos. Os microsserviços são modelados principalmente em torno de domínios de negócio, evitando os diversos problemas da tradicional arquitetura em camadas.

2. REVISÃO TEÓRICA

Este item apresenta os conceitos e fundamentação teórica dos assuntos abordados neste trabalho.

2.1 ARQUITETURA MONOLÍTICA

Em engenharia de software, a arquitetura Monolítica é um estilo arquitetural onde a interface com o usuário e camada de acesso a dados são combinados em um único programa. O código do sistema é centralizado em uma única base de código, que quando compilada, gera um único artefato, tornando o desenvolvimento e testes mais simples, quando comparado a outros estilos arquiteturais.

O código fonte pode ser bem estruturado, aplicando conceito de classes e pacotes, podendo ser dividido em componentes, mas fortemente acoplados.

As principais linguagens de programação oferecem abstrações para quebrar a complexidade da aplicação em módulos. Entretanto, são projetadas para a criação de um único executável monolítico, no qual toda a modularização utilizada é executada em uma mesma máquina. Portanto, os módulos compartilham os mesmos recursos computacionais, tais como processador, memória, disco, rede, bancos de dados e sistema de arquivos.

Mesmo para aplicação monolítica modularizada, há alto grau de dependência entre componentes e módulos do sistema, onde a inclusão ou manutenção de funcionalidades podem gerar inconsistências, se não forem devidamente planejadas e executadas.

A escalabilidade em sistemas monolíticos é simples e fácil de implementar, já que basta replicar o programa em sua totalidade. Entretanto, mesmo que apenas parte do sistema, recurso ou funcionalidade precise de uma nova instância, a arquitetura

exige que todo sistema seja replicado, aumentando custos e uso de recursos computacionais de forma irregular e mal otimizada.

A centralização de código fonte cria uma alta dependência sobre a linguagem de programação escolhida, mesmo que ela não faça mais sentido dentro do contexto do sistema ou esteja tecnologicamente defasada.

A publicação de aplicações monolíticas em ambiente de produção exige que todas as funcionalidades do sistema fiquem fora do ar, até a completa atualização do sistema. Existe um risco operacional e de negócio considerável, principalmente para sistemas de alta criticidade, além de um esforço maior para os times de desenvolvimento, análise e testes a fim de garantir que a aplicação atenda todos os requisitos funcionais, sejam eles novos ou já existentes.

A curva de aprendizado do time de desenvolvimento para sistemas monolíticos é relativamente maior quando comparado com outros estilos arquiteturais. Isso se deve alto grau de acoplamento do sistema, onde a manutenção de código exige maior conhecimento do código fonte.

2.2 Computação Orientada a Serviços

A computação orientada a serviços representa uma nova geração da plataforma da computação distribuída. Como tal, ela abrange muitas coisas, incluindo seu próprio paradigma de design e princípios de design, catálogos de modelos de design, linguagem padrão, um modelo arquitetônico distinto, conceitos, tecnologias e frameworks relacionados (Erl, 2009).

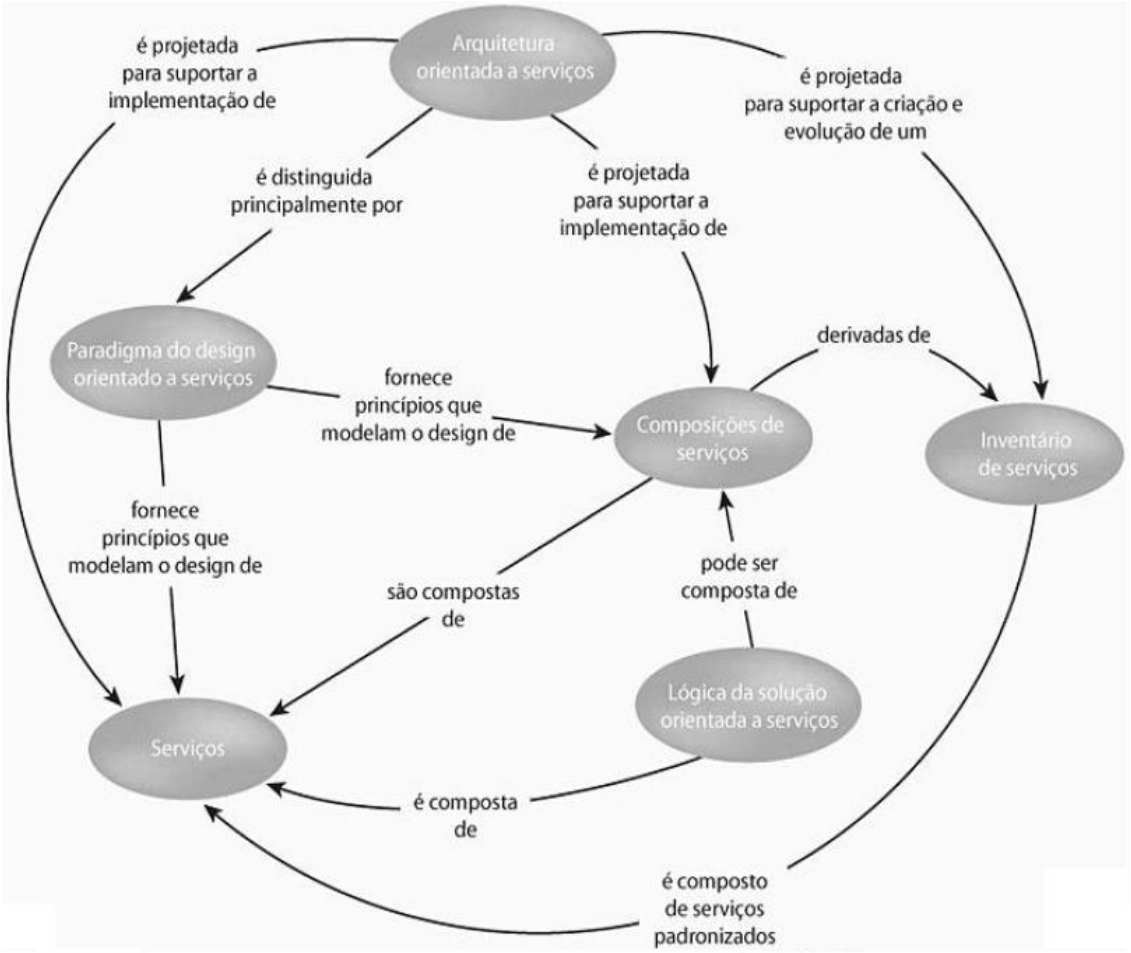


Figura 1: Computação Orientada a Serviços

Fonte: Erl (2009)

2.2.1 SERVIÇO

O nome “serviço” é definido no dicionário como “O desempenho de trabalho (uma função) por alguém para outro”. Contudo, serviço, como o termo é geralmente entendido, também combina as ideias relacionadas com (Mackenzie, et al., 2011):

- A competência de executar o trabalho para outro;
- A especificação do trabalho oferecido para outro;
- A oferta para executar trabalho para outro.

Um serviço é qualquer ato ou desempenho que uma parte pode oferecer a outra e que seja essencialmente intangível e não resulta na propriedade de nada. Sua produção pode ou não estar vinculada a um produto físico (Kotler,1995).

Em artigo intitulado “ABC da SOA” um serviço pode ser representado como alguma funcionalidade ou tarefa, podendo ser compartilhado e reutilizado em diversas áreas da empresa (ABC da SOA, 2010).

2.2.2 WEB SERVICE

Na década de 1990, o desenvolvimento web revolucionou a troca de informações organizacionais. Os computadores clientes poderiam acessar informações de servidores geograficamente distribuídos de suas próprias organizações. No entanto, o acesso era exclusivamente por meio de navegadores (*browsers*) e o acesso direto as informações não era muito prático. (Sommerville, 2011). Para contornar esse problema, foi proposto o web service. Ele é a ideia mais abstrata de um serviço, uma aplicação que fornece um recurso computacional ou informações para outras aplicações.

Segundo Sommerville (2011, p.354), um web service é: “uma representação-padrão para algum recurso computacional ou de informações que pode ser usado por outros programas”. Um web service é uma instância de uma ideia mais geral de um serviço, que é definido segundo Sommerville (2011, p.355 apud Lovelock et. al.,1996) como:

Um ato ou desempenho oferecido de uma parte para outra. Embora o processo possa ser vinculado a um produto físico, o desempenho é essencialmente intangível e normalmente não resulta na posse de qualquer um dos fatores de produção.

No contexto de aplicações, podemos dizer que um web service disponibiliza uma ou mais funcionalidades para outras aplicações, funcionalidades que por sua vez podem ser um simples acesso a dados ou até mesmo o processamento de alguma tarefa.

As organizações que desejam disponibilizar funcionalidades para outros programas podem fazê-lo definindo e publicando uma interface de web service. A interface também chamada de contrato do serviço define quais funcionalidades estão disponíveis e como elas deverão ser acessadas.

Os serviços devem ser construídos para serem reutilizáveis, onde a mesma funcionalidade possa ser compartilhada entre várias aplicações. Para que os serviços estejam visíveis e possam ser usados por diferentes aplicações, eles devem ser independentes de linguagens de programação, plataformas ou sistemas operacionais, característica que denominamos “baixo acoplamento”. Além disso, eles devem se comunicar usando um padrão aplicável por todos.

O acoplamento é uma medida qualitativa do grau de dependência entre componentes, de modo que, quanto maior a dependência, maior será o acoplamento (Pressman, 2009).

2.2.3 Arquitetura Orientada a Serviços

Segundo AALST (2006), a Arquitetura Orientada a Serviços – (SOA) é vista como uma das principais tecnologias para permitir flexibilidade e reduzir a complexidade de sistemas computacionais.

Erl (2009, p.24) afirma que um dos benefícios mais importantes que SOA fornece é a melhora da produtividade e tempo de resposta das empresas frente ás necessidades do negócio. Nas palavras do autor:

“A SOA estabelece um modelo arquitetônico que visa a aprimorar a eficiência, a agilidade e a produtividade de uma empresa, posicionando os serviços como os principais meios para a solução lógica seja representada no suporte à realização dos objetivos estratégicos associados à computação orientada a serviços”.

SOA é um paradigma de desenvolvimento de sistemas distribuídos que permite integrar aplicações independentemente de linguagens de programação, plataformas ou sistemas operacionais, onde o elemento chave é o “serviço”.

Os serviços são implementados por *Web Services,* construídos aplicando protocolos-padrão, tais como XML, SOAP E WSDL. Uma vez que os padrões design, mensagens e contratos estão padronizados, é possível estabelecer uma arquitetura orientada a serviços. Os principais padrões da arquitetura orientada a serviços são (Sommerville, 2011):

1. SOAP. Esse é o padrão para troca de mensagens que oferece suporte à comunicação entre os serviços;
2. WSDL. A Linguagem de definição da interface do serviço, ou seja, do contrato;
3. WS-BPEL. É um padrão para linguagens de *workflow*, que é usada para definir programas de processo que envolve vários serviços diferentes.

Note que embora o SOA seja comumente implementado usando *Web Services*, os serviços podem ser implementados por outras estratégias de implementação. As arquiteturas e as tecnologias baseadas em *Web Services* são específicas e concretas. Enquanto os conceitos no Modelo de Referência se aplicam a estes sistemas, os *Web Service*s são também soluções específicas a serem parte de um modelo de referência genérico. (Mackenzie, Laskey et al., 2011).

SOA possui as melhores práticas para integração de aplicações, fazendo uso de conceitos que foram utilizados com sucesso em implementações baseadas em componentes e integrações de aplicativos coorporativos. (Keen, Bond, *et al.*, 2005).

Sommerville resume o potencial das abordagens orientadas a serviços Sommerville (2011, p.358 apud Newcomer e Lomow, 2005):

Impulsionada pela convergência de tecnologias-chave e a adoção universal de web services, a empresa orientada a serviços promete melhorar significativamente a agilidade empresarial, a velocidade da inserção de novos produtos e serviços no mercado, reduzir custos e melhor a eficiência operacional.

Como forma de arquitetura de tecnologia, uma implementação SOA pode consistir em uma combinação de tecnologias, produtos, *Application Programming Interface - (*API), extensões da infraestrutura de suporte várias outras partes.

A implementação da arquitetura orientada a serviços é exclusiva em cada empresa; contudo, ela é caracterizada pela introdução de novas tecnologias e plataformas que suportam especificamente a criação, a execução e a evolução das soluções orientadas a serviços. (Erl, 2009).

2.2.4 Diferença Entre SOA e Web Service

SOA é a arquitetura abrangente para criar aplicações dentro de uma empresa, onde os programas são criados aplicando uma metodologia de desenvolvimento de software específica, conhecida como programação orientada a serviço. Web services são um conjunto de mecanismos-padrão de comunicação criados sobre a *World Wide Web*. Ou seja, o *Web Service* é uma metodologia para conectar e comunicar. Enquanto SOA é uma estratégia de TI (CIO, 2015).

2.3 Microservices

Microsserviços também conhecido como arquitetura de microsserviços é um estilo arquitetural que estrutura um aplicativo como uma coleção de serviços que possuem as seguintes características:

1) Altamente sustentável e testável
2) Alta granularidade
3) Fracamente acoplada
4) Criado em torno do domínio de negócio
5) Independentemente implantável
6) Time de desenvolvimento pequeno

O que diferencia a arquitetura de microsserviços das abordagens monolíticas tradicionais é como ela decompõe a aplicação por funções básicas. Cada função é denominada um serviço e pode ser criada e implantada de maneira independente. Isso significa que cada serviço individual pode funcionar ou falhar sem comprometer os demais.

A arquitetura de microsserviços promove a criação de serviços granulados com seus próprios ciclos de Vida, que colaboram juntos. Os microsserviços são modelados principalmente em torno de domínios de negócio, evitando os diversos problemas da tradicional arquitetura em camadas (Sam, Newman).

A arquitetura de Microsserviços tem como objetivo solucionar algumas restrições impostas pela arquitetura Monolítica tradicional e melhorar o cumprimento de requisitos não funcionas relacionados a disponibilidade, escabilidade, tolerância/recuperação a erros e desempenho.

Os serviços são independentes e trabalham com seu próprio processo, ou seja, cada serviço roda em processo isolado dentro do sistema operacional, sem compartilhamento de recursos computacionais. Esta característica promove a alta escabilidade, excelência no gerenciamento de recursos computacionais e a alta disponibilidade da aplicação, portanto, caso um ou mais serviços da coleção estiverem inoperantes, não afetará a execução dos demais serviços.

Os serviços possuem baixo acoplamento e interdependência entre si, podendo combinar diferentes tecnologias, onde a escolhe da linguagem de programação poderá ser feita de acordo com necessidades de projeto.

Os serviços geralmente entregam pequenas funcionalidades, tornando o desenvolvimento mais simples. Entretanto, por ser uma aplicação distribuída, a comunicação entre serviços, alta gama de componentes e governança tendem a se tornarem complexas.

Um sistema na formal de microsserviços poderá se tornar altamente complexo, dependendo da composição e disposição dos serviços, e exigirá um investimento considerável na fase de testes.

O monitoramento dos serviços se torna uma tarefa crucial para garantir o perfeito funcionamento do ecossistema no intuito de medir possível pontos de gargalo e falhas.

A publicação de aplicações na forma de microsserviços poderá ser feita serviço a serviço, de forma isolada, mantendo o funcionamento dos demais serviços do sistema. Esta característica promove um menor risco operacional e de negócio, principalmente para sistemas de alta criticidade, onde serviços essenciais poderão continuar em funcionamento.

A implementação de uma arquitetura de microsserviços exige alguns pré-requisitos para garantir o máximo de aproveitamento do ecossistema. Para melhor desempenho e escabilidade é altamente recomendável executá-los em ambiente de nuvem. Tal medida pode implicar em maior custo de projeto, já que será necessário destinar um valor mensal para hospedar os serviços na nuvem.

Segundo Martin Fowler, autor renomado no tema de arquitetura de software, um sistema complexo que utiliza microsserviços tem um custo de manutenção menor, comparado com aplicações monolíticas.

2.4 DIFERENÇAS ENTRE MONOLÍTICO E SOA

Se você conhece o método de decompor aplicações em funções essenciais para evitar os problemas provocados pelas arquiteturas monolíticas é porque o estilo de arquitetura de microsserviços é semelhante ao da arquitetura orientada a serviço (SOA), já consagrado no desenvolvimento de programas de software.

Nos primórdios do desenvolvimento de aplicações, até mesmo as alterações mais insignificantes em uma aplicação pronta exigiam uma atualização da versão de atacado, com um ciclo próprio de garantia da qualidade (QA). Isso, provavelmente, atrasava o trabalho de muitas subequipes. Muitas vezes, essa abordagem é chamada de "monolítica" porque o código-fonte da aplicação toda era incorporado em uma única unidade de implantação, como .war ou .ear. Se a atualização de alguma das partes causasse erros, era necessário desativar a aplicação inteira, reverter a escala e corrigir o problema. Embora essa abordagem ainda seja viável

para aplicações menores, as empresas em ampla expansão não podem se dar ao luxo de sofrer com tempo de inatividade.

A arquitetura orientada a serviço serve pra resolver essa questão, pois estrutura as aplicações em serviços distintos e reutilizáveis que se comunicam por meio de um Enterprise Service Bus (ESB). Nessa arquitetura, os serviços individuais, cada um deles organizado em torno de um processo de negócios específico, aderem a um protocolo de comunicação, como SOAP, ActiveMQ ou Apache Thrift, para que sejam compartilhados por meio do ESB. Quando reunidos em um pacote integrado por meio de um ESB, esses serviços formam uma aplicação.

Por um lado, isso permite criar, testar e ajustar os serviços de maneira simultânea, eliminando os ciclos de desenvolvimento monolíticos. No entanto, por outro lado, o ESB representa um ponto único de falha no sistema inteiro. Portanto, todo o esforço empregado para eliminar uma estrutura monolítica, de certo modo, serviu apenas para criar outra: o ESB, que potencialmente pode congestionar toda a organização.

2.5 DA SOA AOS MICROSERVICES

Os microsserviços podem se comunicar entre si, normalmente de maneira stateless. Portanto, as aplicações criadas dessa maneira podem ser mais tolerantes a falhas e depender menos de um único ESB. Além disso, as equipes de desenvolvimento podem escolher as ferramentas que desejarem, pois os microsserviços podem se comunicar por meio de interfaces de programação de aplicações (APIs) independentes de linguagem.

Levando em consideração a história da SOA, os microsserviços não são uma ideia completamente nova. Porém, eles se tornaram mais viáveis graças aos avanços nas tecnologias de conteinerização. Com os containers Linux, agora é possível executar várias partes de uma aplicação de maneira independente no mesmo hardware e com um controle muito maior sobre os componentes individuais e ciclos de vida.

Juntamente com as APIs e as equipes de DevOps, os microsserviços em containers são os pilares das aplicações nativas em cloud.

3. DESENVOLVIMENTO

NESTA SEÇÃO SERÃO APRESENTADAS AS ESCOLHAS QUE CONDUZIRAM O DESENVOLVIMENTO DO PROJETO, BEM COMO AS JUSTIFICATIVAS PARA AS MESMAS.

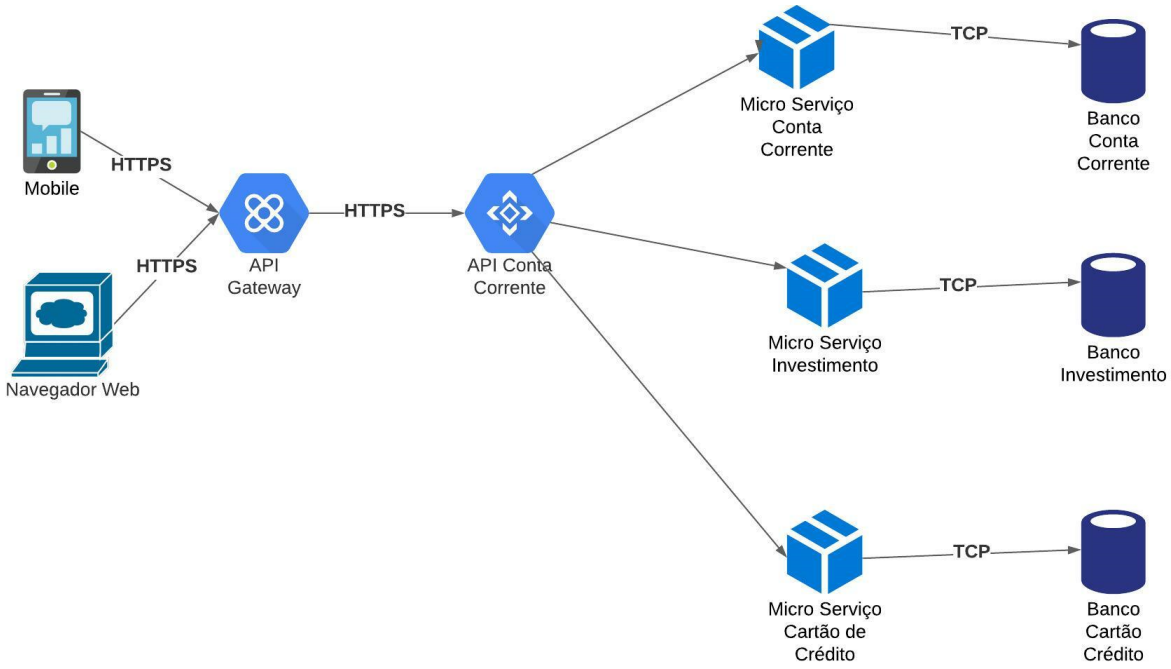
3.1 TECNOLOGIAS/FERRAMENTAS

O projeto foi desenvolvimento usando a Quarkus, um framework Java nativo em Kubernetes. Por ser projetado para desenvolver microsserviços em nuvem, o Quarkus possui alguns recursos nativos de resiliência se tornando a escolha ideal.

O uso do OEM Hibernate possui mapeamento relacional das entidades, agilizando consideravelmente a programação além de proporcionar a criação de tabelas, relacionamentos e campos automaticamente. Não é necessário criar comandos SQL, pois o framework se encarrega desta tarefa.

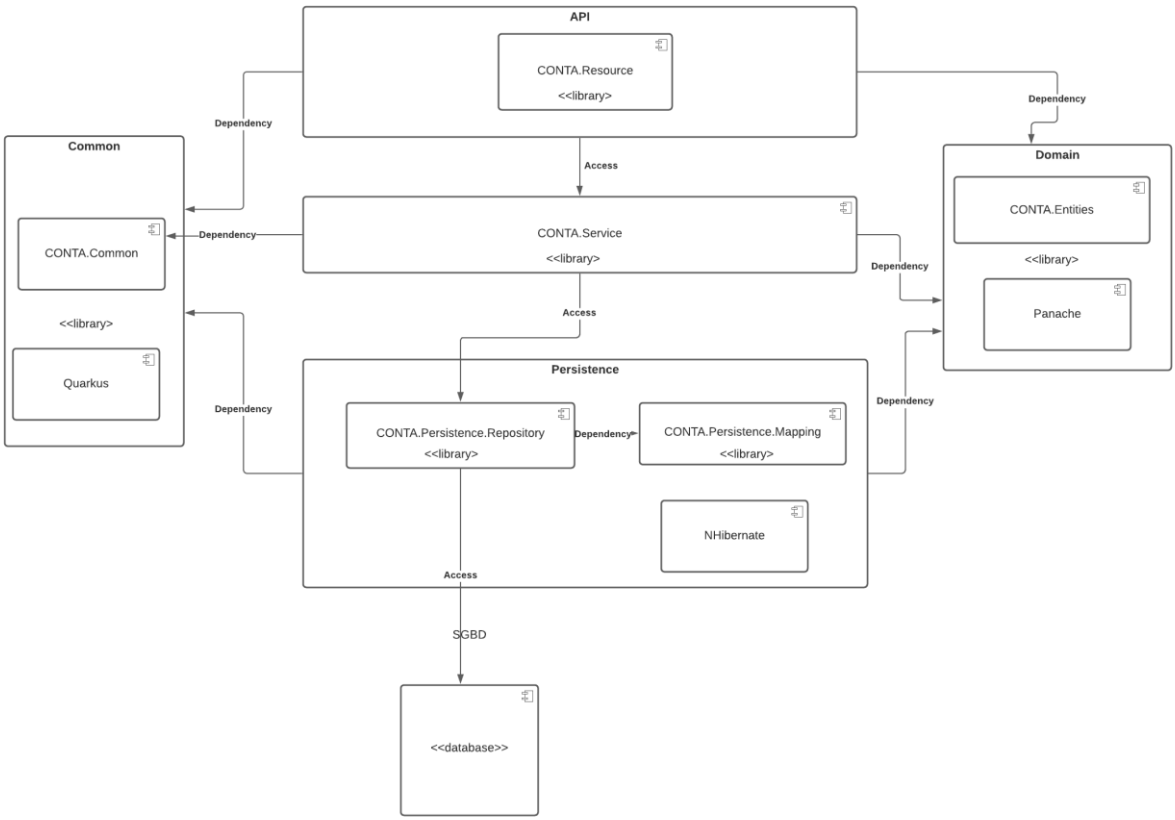
3.2 VISÃO MACRO

Segue representação macro da arquitetura do sistema:



3.3 VISÃO IMPLEMENTAÇÃO

Os três serviços (Conta corrente, investimentos e cartão de crédito) foram criados com a mesma arquitetura base:



3.3.1 RESOURCE

A camada de controladores tem como objetivo expor os métodos da API para camada de apresentação. Nesta camada são definidos os nomes dos métodos, parâmetros de entrada e saída e verbos HTTP (GET, PUT, POST e DELETE).

3.3.2 SERVICE

A camada de domínio tem como objetivo concentrar todas as regras de negócio. Todas as operações relacionadas a cada entidade do sistema bem como suas interações serão orquestradas através desta camada.

3.3.3 ENTITIES

Esta biblioteca contém as entidades da aplicação que representam seu domínio. Elas serão utilizadas para realização das regras de negócio e comunicação com o banco de dados.

3.3.4 PERSISTENCE

Todo o acesso com o intuito de recuperação ou persistência de dados será feito pelos componentes desta camada. Sendo assim, todo o acesso ao banco de dados será encapsulado das demais camadas como investimento em adaptabilidade e facilidade de manutenção do software.

Esta camada deverá ser projetada para prover o menor acoplamento possível com o banco de dados com o qual se conecta. Esta camada é dividida em repositórios e mapeamento. A camada de repositório armazena métodos de consulta e persistência de dados. A camada de mapeamento faz o mapeamento relacional entre as classes de negócios e estrutura do banco de dados. Exemplo: realiza a ligação entre determinado atributo de uma classe com campo de uma tabela.

3.3.5 COMMON

Nesta biblioteca, serão colocados todos os recursos e objetos da aplicação que não tiverem um fim arquitetural e restrições associadas, e que sirvam somente como ajudantes (helpers) para os demais componentes da arquitetura.

Exemplos destes objetos são validadores, conversores e constantes.

3.4 REQUISITOS NÃO FUNCIONAIS

Nesta seção são detalhados os requisitos não funcionais relacionados à:

1) Segurança
2) Escabilidade
3) Monitoramento
4) Consistência de dados
5) Disponibilidade
6) Testabilidade
7) Monitoramento

3.4.1 SEGURANÇA

Foi aplicado o uso da extensão Quarkus OpenID Connect (OIDC) para proteger seus aplicativos JAX-RS usando Bearer Token Authorization onde Bearer Tokens são emitidos por OpenId Connect e servidores de autorização compatíveis com OAuth 2.0, como Keycloak.

A Autorização de Token de Portador é o processo de autorização de solicitações HTTP com base na existência e validade de um Token de Portador que fornece informações valiosas para determinar o assunto da chamada, bem como se um recurso HTTP pode ou não ser acessado.

3.4.2 ESCALABILIDADE

A plataforma Quarkus permite a publicação de microsserviços em nuvem, com Kubernetes, garantindo a criação de múltiplas instâncias da aplicação. Para critério de exemplificado, no escopo do trabalho, os serviços de conta corrente, cartão de crédito e investimentos rodarão em Docker.

3.4.3 TESTABILIDADE

Uso de testes unitários fornecidos pela biblioteca Fault Tolerance, podendo aplicar diversos testes.

3.4.4 DISPONIBILIDADE

A plataforma Quarkus permite a publicação de microsserviços em nuvem, com Kubernetes, garantindo a criação de múltiplas instâncias da aplicação, garantindo a alta disponibilidade. Para critério de exemplificado, no escopo do trabalho, os serviços de conta corrente, cartão de crédito e investimentos rodarão em Docker.

3.4.5 MONITORAMENTO

Foi aplicado o uso do MicroProfile Metrics permite que os aplicativos reúnam várias métricas e estatísticas que fornecem informações sobre o que está acontecendo dentro do aplicativo.

As métricas podem ser lidas remotamente usando o formato JSON ou o formato OpenMetrics, para que possam ser processadas por ferramentas adicionais, como o Prometheus, e armazenadas para análise e visualização.

Exemplo de caminho: http://localhost:8080/q/metrics

3.4.6 CONSISTÊNCIA DOS DADOS

Cada serviço possui um banco de dados dedicado.
As operações encadeadas são controladas por transações:

```
Fatura faturaEntity = new Fatura();
try {
        transaction.begin();
        faturaEntity = faturaService.addFatura(fatura);
        cartaoService.aumentaLimiteEmUso(fatura.getCartao().id, fatura.getValor());
        transaction.commit();
    }
    catch(Throwable e) {
        // do something on Tx failure
        transaction.rollback();
    }
```

3.4.7 RESILIÊNCIA

Um dos desafios trazidos pela natureza distribuída dos microsserviços é que a comunicação com sistemas externos não é inerentemente confiável.

Isso aumenta a demanda por resiliência de aplicativos. Para simplificar a criação de aplicativos mais resilientes, o Quarkus fornece SmallRye Fault Tolerance, uma implementação da especificação MicroProfile Fault Tolerance:

1)Timeout
2)Fallback
3)Retrye
4)CircuitBreaker

Exemplo de implementação:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Retry(maxRetries = quantidadeRetry, delay = delayValue, delayUnit = ChronoUnit.SECONDS)
@Timeout(timeout)
@Fallback(fallbackMethod = "fallbackContaCorrente")
@CircuitBreaker(requestVolumeThreshold = requestVolumeThresholdValue)
public List<ContaCorrente> listContaCorrente(){
```

3.5 Requisitos funcionais

O escopo do trabalho abrange a execução de três microsserviços, a conta corrente, investimentos e cartão de crédito.

3.5.1 Conta Corrente

A conta corrente fornece operações de criação de conta, alteração de dados cadastrais e fechamento da conta.

A conta corrente possui os serviços de **débito** e **crédito** que são disponibilizados para microsserviços de investimentos e cartão de crédito.

3.5.2 Investimento

Quando é criado um investimento, é executada a transação de débito do microsserviço de conta corrente. Não é possível alterar investimento depois de criado.

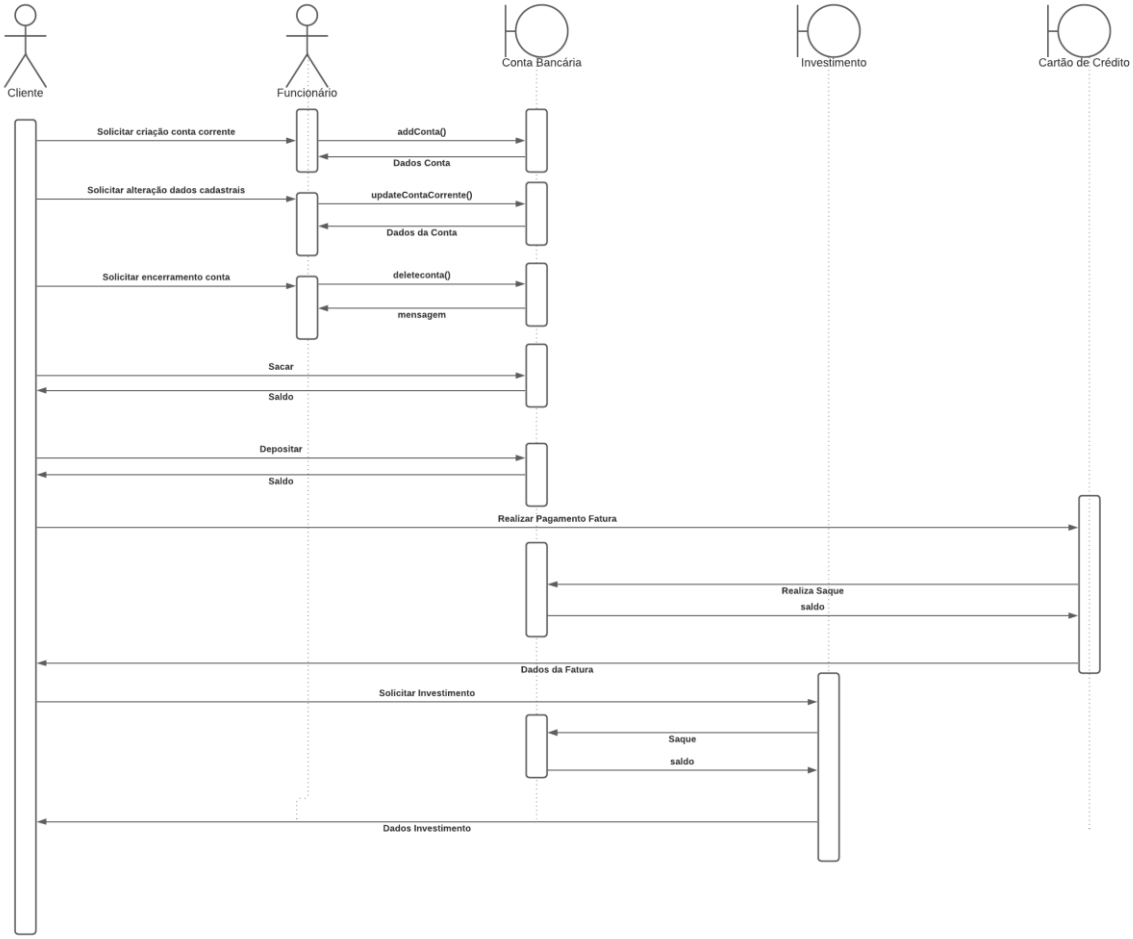
3.5.3 Cartão de Crédito

O microsserviço de cartão de crédito possui três entidades, **cartão de crédito, fatura e pagamento**. Um cartão de crédito pode possuir uma ou mais faturas. Cada fatura pode ter um ou mais pagamentos.

Sempre que uma fatura é paga, é executada a transação de débito do microsserviço de conta corrente. O limite do cartão é liberado após pagamento.

Sempre fatura é gerada, o limite do cartão de crédito é retido.

3.6 DIAGRAMA DE SEQUÊNCIA



CONSIDERAÇÕES FINAIS

Este trabalho proporcionou um melhor entendimento da arquitetura de microsserviços, conhecimento os pontos positivos e negativos desta arquitetura, além de saber em quais situações devemos ou não usá-la. Foi possível ter uma clara definição das diferenças arquiteturais entre microsserviços, webservices e arquitetura monolítica.

O Quarkus é um excelente framework, com muitos recursos além de ser bem ativo na comunidade, com muitos tutoriais, códigos e exemplos. A geração da aplicação em Docker se tornou mais simples, com os recursos proporcionados por este framework.

Acreditamos que o esforço empregado neste trabalho além de proporcionar muito conhecimento, também servirá como referência para o trabalho de conclusão de curso.

REFERÊNCIAS BIBLIOGRÁFICAS

Roger Pressman – Engenharia de Software. Tradução de Ariovaldo Griesi e Mario Moro Fecchio. 7. ed – São Paulo: AMGH, 2011. ISBN 978-85-63308-33-7.

Ian Sommerville – Engenharia de Software. Tradução de Ivan Bosnic. 9. ed – São Paulo: Pearson Education, 2011. ISBN 978-85-7936-108-1.

Modelo de maturidade SOA – The Open Group. 2015. Disponível em <http://www.opengroup.org/content/osimm-how-measure-success-soa-and-design-roadmap>. Acesso em: outubro de 2015.

The Open Group. 2015. Disponível em: <http://blog.opengroup.com.br>. Acesso em: outubro de 2015.

W3 ORG SOAP version 1.2. 2000. Disponível em: <http://www.w3.org/TR/soap>. Acesso em: setembro de 2015.

Oasis Web Services Reliable Messaging. 2006. Disponível em: <http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-spec-cd-04.html>. Acesso em: setembro de 2015.

IBM Entendendo Especificações de Serviços da Web. 2011. Disponível em: <http://www.ibm.com/developerworks/br/webservices/tutorials/ws-understand-web-services4>. Acesso em: outubro de 2015.

OSIMM - Modelo de Maturidade SOA Aberto. 2015. Disponível em: <http://sensedia.com/blog/soa/osimm-modelo-de-maturidade-soa-aberto/>. Acesso em: outubro de 2015.

Josuttis, N. M. SOA na Prática. Tradução de Ivan Bosnic. 1. ed. Rio de Janeiro: Alta Books, 2008. 265 p. ISBN 978-85-7608-184-5.

Drew, D.W. “Tailoring the software engineering Institute's (SEI) Capability Maturity Model (CMM) to a software sustaining engineering organization”. Proceedings of Conference on Software Maintenance, 1992.

ABC da SOA. CIO. 2006. Disponível em: <http://cio.com.br/tecnologia/2006/07/17/idgnoticia.2006-07-17.3732358054/>. Acesso em: novembro de 2015.

Mackenzie, C. et al. Reference Model for Service Oriented Architecture 1.0. OASIS – Advancing Open Standards for the Informantion Society, 2006. Disponível em: <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>. Acesso em: outubro 2015.

Erl, Thomas. SOA Princípios de design de serviços. São Paulo: Pearson Prentice Hall, 2009.

Gartner Group. The 13 Most Common SOA Mistakes and How to Avoid Them. September, 2009.

AALST et al. “*SOA-Based Architecture Framework*”. 2006. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2006/827/>. Acesso em: novembro de 2015.

Kotler, Philip. Administração de Marketing: análise, planejamento, implementação e controle. Tradução Ailton Bomfim Brandão; 4ª ed., São Paulo, Atlas, 1995.

Keen, M. et al. Patterns: Implementing an SOA Using an Enterprise Service Bus. IBM RedBooks, Disponível em: <http://www.redbooks.ibm.com/abstracts/sg246773.html?Open>. Acesso em: outubro 2015.

Bieberstein, N. et al. Executing SOA - A Pratical Guide for the Service-Oriented Architect. 1. ed. Boston: IBM Press, 2008. 213 p. ISBN 978-0-13-235374-8.

Soluções de SOA. Oracle Suite. 2015. Disponível em: <http://iprocess.com.br/tecnologias/oracle-soa-suite/>. Acesso em: setembro 2015.

Oracle SOA Suite. 2013. Disponível em: <http://www.oracle.com/br/products/middleware/soa/overview/index.html>. Acesso em: setembro de 2015.

Greg Flurry. IBM. Enterprise Service Bus. 2007. Disponível em: <http://www.ibm.com/developerworks/webservices/library/ar-esbpat1>. Acesso em: novembro de 2015.

Oracle Web Logic. 2013. Disponível em: <http://www.oracle.com/us/products/middleware/application-server/oracle-weblogic-server-ds-1391360.pdf>. Acesso em: novembro 2015.

Orenstein, David. Application Programming Interface. 2000. Disponível em: <http://www.computerworld.com/article/2593623/app-development/application-programming-interface.html>. Acesso em novembro 2015.

Agarwal, Sachin. API versus SOA. Are they different. 2013. Disponível em: <https://blog.akana.com/api-vs-soa-different/>. Acesso em dezembro 2015.

Chris, Richardson. Monolithic Architecture. 2019. Disponível em: https://microservices.io/patterns/monolithic.html

Chris, Richardson. Microservice architecture. 2019. Disponível em: https://microservices.io/

Sam Newman, Building Microservices 2019

Red Hat. What are microservices. Disponível em: https://www.redhat.com/pt-br/topics/microservices/what-are-microservices