FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" – UNIVEM
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

LEONARDO ROSS TORQUATO LIMA

IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS

MARÍLIA
2015

LEONARDO ROSS TORQUATO LIMA

IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS

Trabalho de Curso apresentado ao Curso de Graduação em Sistemas de Informação, do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Ms. Ricardo José Sabatine.

MARÍLIA
2015

LIMA, Leonardo Ross Torquato
IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS / Leonardo Ross Torquato Lima; orientador: Prof. Ms. Ricardo José Sabatine. Marília, SP: [s.n.], 2015.

Trabalho de Conclusão de Curso (TCC) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha.

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
MANTIDO PELA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Leonardo Ross Torquato Lima

TÍTULO: IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS.

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Sistemas de Informação.

Nota: 10 (Dez)

Orientador: Ricardo José Sabatine Ricardo Sabatine

1º.Examinador: Ildeberto de Gênova Bugatti

2º.Examinador: Fabio Lucio Meira

Marília, 30 de novembro de 2015.

Dedico este trabalho à minha família,
que sempre me incentivou a seguir
meus sonhos e confiou nas minhas
escolhas.
À minha namorada, por todo o amor e
carinho durante nossa jornada.

AGRADECIMENTOS

À Deus, por todos os desafios que enfrentei e vitórias que conquistei.

À minha namorada e à sua família, pelo apoio durante estes anos.

Ao meu professor e orientador Prof. Ms. Ricardo José Sabatine, por todo o conhecimento compartilhado e por sempre me incentivar a melhorar.

À Prof. Ms. Giulianna Marega Marques e à toda a equipe da Persys pelo aprendizado constante.

Aos amigos Evandro, Rodrigo e Thiago, pelo companheirismo e por tornar a graduação mais divertida.

À todos os professores do curso e toda a turma de Sistemas de Informação, por contribuir para minha formação acadêmica.

E à todos aqueles que fizeram parte da minha vida e que contribuíram para eu me tornar quem sou.

“If I have seen further, it is by standing on the shoulders of giants”

Isaac Newton

LIMA, Leonardo Ross Torquato. **IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS**. 2015. 76f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2015.

RESUMO

Durante o ciclo de vida de aplicações corporativas e arquiteturas que sustentam suas funcionalidades, desenvolvedores passam por desafios semelhantes, apesar de estarem focados em soluções com finalidades e experiências diferentes. O objetivo deste trabalho é a implementação de uma arquitetura baseada em microserviços, que visa simplificar o ciclo de vida de uma aplicação. Esta implementação deve tirar proveito das principais vantagens que este modelo arquitetônico oferece como fácil manutenção, escalabilidade natural e tolerância à falhas. Como resultados deste trabalho, foi desenvolvida uma aplicação que é constituída por vários microserviços e um aplicativo móvel que consome as funcionalidades disponibilizadas pela aplicação.

Palavras-chave: Arquitetura, micro serviços, manutenção, escalabilidade, reusabilidade.

LIMA, Leonardo Ross Torquato. **IMPLEMENTAÇÃO DE UMA ARQUITETURA BASEADA EM MICROSERVIÇOS**. 2015. 76f. Trabalho de Curso (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2015.

ABSTRACT

During the lifecycle of enterprise software and the underlying architecture that supports it's functionalities, software developers often face similar challenges, even though they are focused in products that have completely different purposes and experiences. The goal of this project is to implement a microservice-based architecture aimed to simplify an application's lifecycle. This implementation must profit from the main advantages offered by this architectural style, *i.e.*, easy maintenance, natural scalability and high reusability. The result of this work was the development of an application composed by several microservices and a mobile app that consumes the application functionalities.

Keywords: Architecture, microservices, maintenance, scalability, reusability.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
DNS	Domain Name System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	Javascript Object Notation
JVM	Java Virtual Machine
NoSQL	Non relational ou Non Structured Query Language
PaaS	Platform as a Service
REST	Representational State Transfer
SGBD	Sistema de Gerenciamento de Bando de Dados
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
WSDL	Web Services Description Language
XML	Extensible Markup Language

SUMÁRIO

INTRODUÇÃO

Segundo Fowler (2002), aplicações corporativas são um tipo distinto de software que apresentam diversos desafios e complexidades durante seu desenvolvimento. Normalmente este tipo de aplicação envolve o armazenamento de grande quantidade de dados, acesso concorrente aos dados por diferentes pessoas, várias telas de interface diferentes, integração com outros sistemas e lógicas de negócio complexas.

Neste contexto, se torna importante a atividade de arquitetura de softwares. Arquitetura é um termo que muito tentam definir, sem chegar num consenso total. Apesar disso, existem dois elementos comuns a essas definições: a divisão no mais alto nível de um sistema e suas partes e decisões que são difíceis de serem alteradas (FOWLER, 2002).

Ao decorrer do tempo, diversos padrões de arquitetura foram adotados para solucionar diferentes problemas de aplicações corporativas como a arquitetura cliente/servidor, arquitetura web e a Arquitetura Orientada a Serviços.

À medida que empresas começaram a migrar suas soluções para nuvem, crescia o sentimento de frustração por parte das equipes de desenvolvimento. Elas precisavam lançar aplicações com arquiteturas que ofereciam suporte para todas suas funcionalidades, mas que também fossem maleáveis o suficiente para enfrentar todos os desafios e adversidades encontrados durante seu ciclo de vida e que estivessem prontas para ser reaproveitadas em outras aplicações no caso de um eventual declínio (TANG, 2013).

Este cenário acabou levando ao estilo arquitetônico de microserviços: uma forma de abordar o desenvolvimento de aplicações como um conjunto de serviços pequenos e independentes. A migração para este estilo arquitetural aconteceu de forma natural para diversas empresas que perceberam que poderiam gerar suas soluções de software de forma mais rápida ao separá-las em pequenas unidades (FOWLER, 2014).

Objetivos

O objetivo deste trabalho é explorar a arquitetura baseada em microserviços a fim de destacar as vantagens e desvantagens que este modelo arquitetônico pode oferecer para desenvolvedores no contexto de aplicações corporativas.

Para isso, será implementada uma solução que possui funcionalidades comuns em aplicações modernas e que utilizará microserviços para compor suas funções.

Além disso tem-se como objetivos específicos:

- Realizar um estudo da evolução das arquiteturas.
- Realizar um estudo sobre arquiteturas baseadas em microserviços.
- Identificar e analisar o processo de negócio do cenário proposto e a decomposição em serviços.
- Identificar o tipo apropriado de composição dos serviços.
- Realizar uma avaliação e identificar tecnologias adequadas para a criação de microserviços.
- Realizar um estudo sobre o desempenho da arquitetura proposta.

Organização do trabalho

Este trabalho está dividido em cinco capítulos.

No primeiro capítulo é apresentada uma contextualização sobre arquiteturas modernas, demonstrando as características dos principais modelos arquitetônicos e sua evolução.

No segundo capítulo é apresentada a arquitetura de microserviços, citando suas principais características, as vantagens e desvantagens de sua utilização, formas de implementá-la e alguns casos de exemplo que tiveram sucesso ao adotá-la.

No terceiro capítulo são descritas as funcionalidades do projeto que foi desenvolvido, como ele foi divido em serviços, as principais tecnologias usadas na implementação e finalmente como os serviços propostos foram concretizados na forma de microserviços.

No quarto capítulo são descritos os resultados obtidos durante a implementação, o aplicativo que foi gerado como resultado do projeto proposto e testes que avaliam como a arquitetura de microserviços que foi implementada se comporta em cenários de alta demanda.

No quinto capítulo é apresentada a conclusão do trabalho desenvolvido e apresenta como ele pode ser melhorado futuramente.

1 ARQUITETURAS E DESIGN DE SOFTWARE

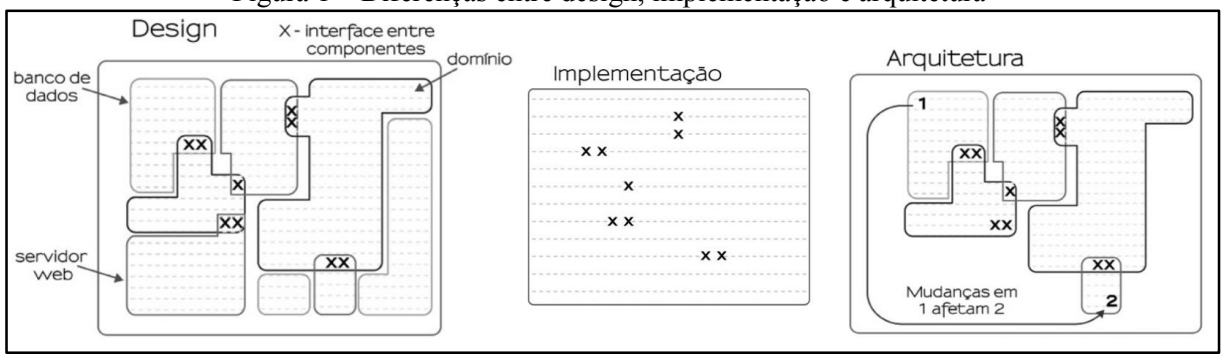
O design de software é uma maneira de interpretar a implementação, analisando e compreendendo as interações entre componentes do sistema, como possíveis impactos que uma mudança em um componente causa em outros componentes que o acessam direta ou indiretamente. Na visão do design, são importantes características das interfaces de comunicação entre partes do sistema e seus componentes em diversos níveis de abstração, desde a comunicação entre classes até a comunicação entre softwares distintos (SILVEIRA, 2012).

Implementação não se restringe ao código escrito por desenvolvedores de uma equipe específica, mas um conjunto de escolhas de ferramentas, versões dos ambientes de desenvolvimento, homologação e produção, o conjunto de linguagens escolhidas, a qualidade do código, entre outras características. Estas escolhas modelam a implementação de um sistema, assim como determinam sua qualidade (SILVEIRA, 2012).

Arquitetura também é um forma de interpretar a implementação, possibilitando analisar e identificar como mudanças em determinados pontos do sistema afetam o sistema como um todo, e de todas as maneiras possíveis. Um exemplo é a mudança em um determinado ponto de uma aplicação fazendo com que este passe a fazer um acesso remoto, em vez de um acesso local. Isto pode afetar diversas partes da aplicação, tornando-a mais lenta. Esta análise não é feita no design, já que este se preocupa com questões mais locais. À visão mais global do sistema, é dado o nome de arquitetura (SILVEIRA, 2012).

Na Figura 1 é ilustrada a diferença entre arquitetura, design e implementação.

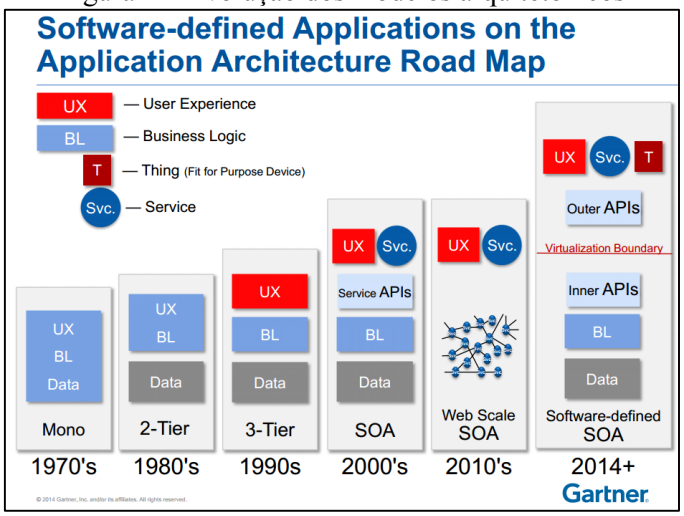
Figura 1 – Diferenças entre design, implementação e arquitetura



Fonte: Silveira, et al, Introdução à Arquitetura e Design de Software, 2012.

Ao decorrer das últimas décadas, diferentes arquiteturas surgiram no contexto de aplicações corporativas. A sua evolução acompanhou o aumento do poder computacional e a disseminação da Internet. Na Figura 2 são representados os modelos arquitetônicos mais populares e como eles evoluíram com o tempo.

Figura 2 – Evolução dos modelos arquitetônicos



Fonte: Gartner, Software defined architecture: Application design for Digital Business, 2014

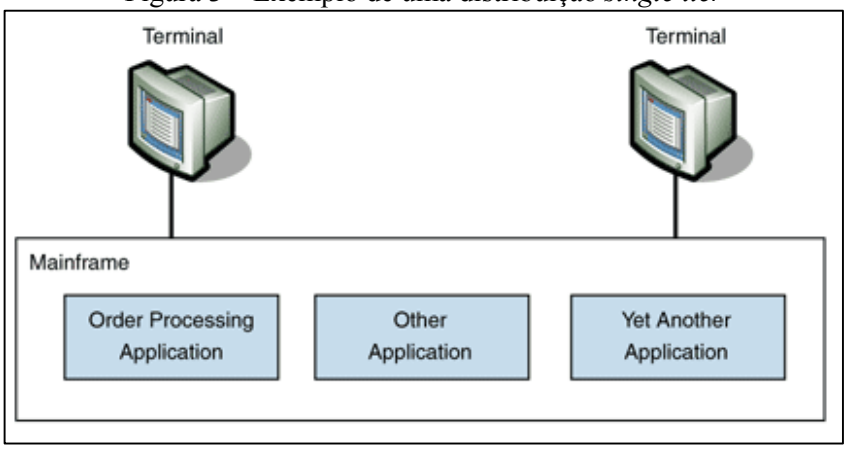
Como poder ser observado na Figura 2, uma característica evolucionária comum entre os modelos arquitetônicos é sua constante divisão em partes menores, com responsabilidades distintas. Isto acontece desde o modelo *Single Tier* que evoluiu para o modelo 2-*Tiers*, até a Arquitetura Orientada a Serviços que evoluiu para a arquitetura baseada em microserviços.

A seguir será apresentada uma visão geral destas arquiteturas.

1.1 Arquitetura *Single Tier*

Segundo a Microsoft Corporation (2003), a arquitetura *Single Tier*, também conhecida como *One-Tier*, consiste em agrupar todas as camadas de uma aplicação como a de apresentação, regras de negócio e armazenamento de dados em um único servidor ou plataforma responsável por todo o processamento. Na Figura 3 é ilustrado como aplicações utilizam a arquitetura single tier.

Figura 3 – Exemplo de uma distribuição *single tier*



Fonte: Microsoft Corporation, Patterns & practices, 2003

Foi o modelo de arquitetura predominante até a década de 1970 devido ao alto custo de aquisição de computadores. Empresas optavam por utilizar um grande servidor central, que seria responsável por centralizar todo o processamento e armazenamento, e diversos terminais que não faziam nenhum processamento, somente ofereciam uma interface para entrada e saída no sistema.

Uma das vantagens oferecidas pelo modelo de arquitetura *single tier* é a simplicidade de *deploy* e administração, já que todos os componentes da aplicação estão concentrados em um único computador. Outra vantagem é a facilidade de garantir a segurança do sistema quando todos usuários estão em uma intranet.

O aumento no número de usuários faz com que a carga do sistema aumente, gerando a necessidade de aumentar a capacidade de memória, armazenamento e processamento do sistema. Eventualmente, a capacidade de escalar verticalmente se esgota, fazendo-se necessária a aquisição de outro servidor. O alto custo de aquisição de mainframes acabava limitando a escalabilidade dos sistemas.

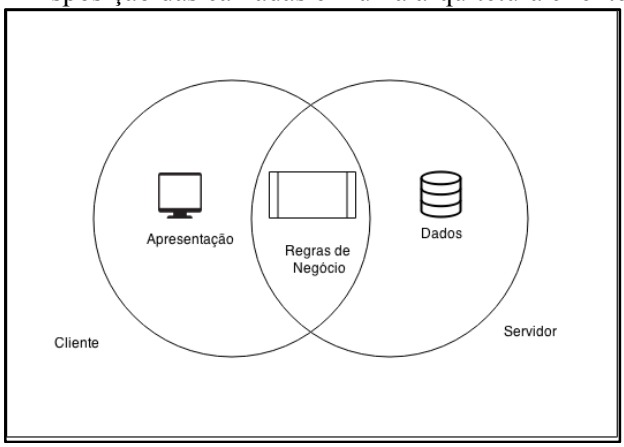
1.2 Arquitetura cliente/servidor

A arquitetura cliente/servidor é um modelo de arquitetura que particiona tarefas ou cargas de trabalho entre o provedor de um recurso ou serviço (servidor) e seus consumidores (clientes) cuja comunicação normalmente ocorre através de uma rede. Um servidor é um host de alta performance que compartilha seus recursos com clientes. Um cliente não compartilha nenhum de seus recursos, mas requisita o conteúdo ou função de um servidor (FARNAGHI, et al, 2009).

Este modelo de arquitetura começou a se popularizar a partir da década de 1980 com a diminuição de custos e aumento do poder de processamento dos computadores, permitindo que a toda a camada de apresentação e parte da lógica de negócio fossem transferidas para os clientes, aliviando parte da carga do servidor.

Segundo Furht (2000), a chave para implantar corretamente esta arquitetura é particionar a complexidade da aplicação, determinando corretamente qual parte residirá no servidor e qual parte será executada no cliente. Isto permite que aplicações maiores e mais escaláveis sejam oferecidas a um número maior de clientes. Na Figura 4 é ilustrada a disposição das camadas em uma arquitetura cliente/servidor.

Figura 4 – Disposição das camadas em uma arquitetura cliente/servidor

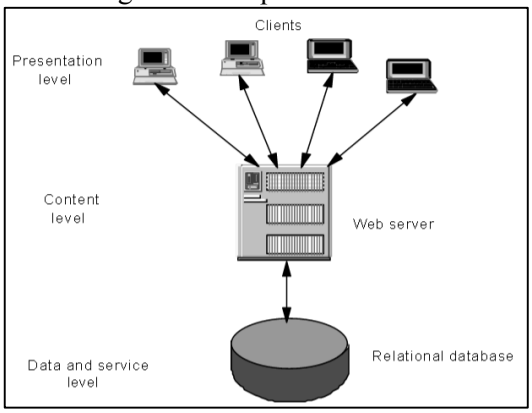


Fonte: Elaborado pelo autor, 2015

1.3 Arquitetura web

A primeira geração de arquitetura de aplicações voltada para a web era baseada na entrega de informações por meio de websites públicos. Esta abordagem, chamada de "*first wave internet*" utiliza a web para disponibilizar informações ao usuário, e permite que ele devolva informações relevantes. O foco primário desse modelo arquitetural é a distribuição em massa de informações públicas por meio da Internet. Essa arquitetura, focada no acesso à informações, consiste em três níveis: apresentação, conteúdo e dados, como ilustrado na Figura 5 (FURHT, et. al., 2000).

Figura 5 – Arquitetura 3 *tiers*

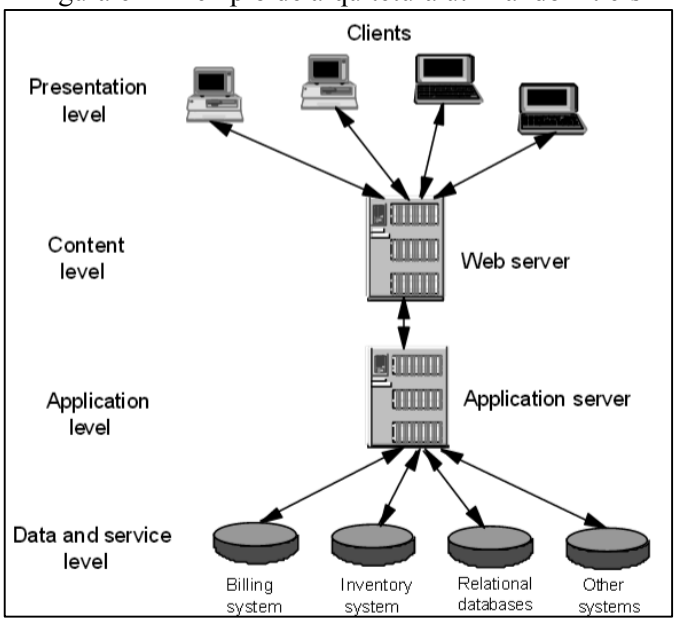


Fonte: Furht, et al, An innovative Internet architecture for application service providers, 2000

No nível de apresentação está o sistema cliente, responsável por exibir as informações de páginas web. O cliente possui componentes de apresentação e de lógica. O *webserver* está no nível de conteúdo, disponibilizando uma visualização interativa de informações de um banco relacional. Finalmente, no nível de dados e serviços existe um banco de dados relacional que fornece informações para o *webserver* (FURHT, et al., 2000).

Com o avanço da Internet, da web e de tecnologias relacionadas, e também da aceitação de padrões de protocolos de comunicação (como o TCP/IP e HTTP), um novo modelo de arquitetura emergiu. Essa arquitetura, também chamada de "*second wave internet*" ou arquitetura de aplicações baseada em redes, tem como foco a distribuição privada de serviços de software específicos através de *intranets* e *extranets*. Nesta arquitetura, além de fornecer informações, as *webpages* também oferecem uma variedade de serviços para acelerar transações de negócios e oferecer serviços adicionais. Esta arquitetura possui n-níveis, como ilustrado na Figura 6, e oferece o máximo de funcionalidade e flexibilidade em um ambiente heterogêneo baseado na web (FURHT, et al., 2000).

Figura 6 – Exemplo de arquitetura utilizando 4 tiers



Fonte: Furht, et al, An innovative Internet architecture for application service providers, 2000

1.4 Arquitetura Orientada a Serviços

Segundo Papazoglou (2003), a Arquitetura Orientada a Serviços (SOA) é uma abordagem ao desenvolvimento de software e infraestrutura de suporte como um conjunto interconectado de serviços, acessíveis através de interfaces e protocolos padronizados de mensagem. Assim que todos os componentes de uma arquitetura empresarial estiverem alocados, aplicações existentes e futuras podem acessar estes serviços conforme a necessidade sem precisar utilizar protocolos ponto a ponto proprietários.

Este estilo arquitetural é particularmente aplicável quando múltiplas aplicações que rodam em diferentes plataforma e tecnologias precisam se comunicar. Desta forma, empresas

podem misturar e combinar serviços para realizar transações de negócio com o mínimo esforço de programação (PAPAZOGLOU, 2003).

1.4.1 Características da Arquitetura Orientada a Serviços

Segundo o World Wide Web Consortium (2004), a Arquitetura Orientada a Serviços é caracterizada através das seguintes propriedades:

Visão lógica: O serviço é uma visão lógica abstrata de programas, bancos de dados, processos de negócio e outros, definido em termos do que ele faz, tipicamente uma operação de negócio.

Orientação a mensagem: O serviço é definido formalmente em relação às mensagens trocadas entre agentes provedores e requisitores, e não das propriedades dos agentes em si. A estrutura interna de um agente, incluindo características como sua linguagem de implementação, estrutura de processos e até a estrutura do banco de dados, são abstraídos deliberadamente na SOA: ao utilizar SOA, não é preciso conhecer como um agente que implementa um serviço é construído. Um benefício chave disso está relacionado aos chamados sistemas legados. Ao evitar qualquer tipo de conhecimento da estrutura interna de um agente, pode-se incorporá-lo a qualquer componente de software ou aplicação que consiga manipular suas mensagens.

Orientação a descrição: um serviço é definido por metadados processáveis por máquinas. A descrição mantém a natureza pública da SOA: somente os detalhes que são expostos ao público e importantes para o uso do serviço devem ser incluídos na descrição. A semântica do serviço deve ser documentada, diretamente ou indiretamente, através de sua descrição.

Granularidade: Serviços tendem a utilizar um número reduzido de operações com mensagens relativamente grandes e complexas.

Orientação a redes: Serviços tendem a ser orientados a utilizar redes, mas isto não é um requisito.

Neutralidade de plataforma: Mensagens são enviadas em um formato padronizado, independente de plataforma, através de suas interfaces.

1.4.2 *Webservice*s tradicionais

Segundo o World Wide Web Consortium (2004), um *webservice* é um software

projetado para suportar de forma transparente a comunicação entre duas máquinas através de uma rede. Ele possui uma interface descrita em um formato compreensível por máquinas (WSDL). Outros sistemas interagem com ele da forma prevista na descrição de sua interface utilizando mensagens SOAP, normalmente transportadas utilizando HTTP com serialização XML em conjunto com outros padrões relacionados à web.

Um *webservice* é uma noção abstrata que deve ser implementada por um agente concreto. O agente é a parte concreta de software ou hardware que envia e recebe mensagens, enquanto o serviço é caracterizado pelo conjunto abstrato de funcionalidades que é oferecido.

1.4.3 *Webservices* baseados em REST

Segundo Fielding (2000), *webservices* que aderem aos padrões REST são chamados de APIs RESTful. APIs RESTful que trabalham via HTTP possuem as seguintes características:

- Possui uma URI base, como por exemplo “http://www.google.com/api”
- Utiliza um tipo de mídia voltado para Internet. A padrão mais popular é o JSON, mas podem ser utilizados outros como XML.
- Implementa os verbos HTTP como forma de comunicação (GET, PUT, POST, DELETE).
- Links de *hypertext* para referenciar o estado.
- Links para referenciar outros tipos de recursos

Na Tabela 1 são relacionados alguns verbos que normalmente são utilizados para implementar uma API RESTful:

Tabela 1 – Verbos HTTP e seu efeito sobre recursos REST

Recurso	GET	PUT	POST	DELETE
URI de coleção, como por exemplo: "http://api.google.com/recursos"	Lista as URIs e talvez os detalhes dos itens de uma coleção.	Substitui a coleção inteira por outra coleção.	Cria um novo item na coleção.	Apaga a coleção inteira.
URI de elemento, como por exemplo: "http://api.google.com/recursos/5"	Recupera um item de uma coleção, representado por meio de um tipo de mídia da Internet.	Substitui o item indicado da coleção, ou o cria se ele ainda não existe.	Trata o item de uma coleção como uma coleção por si só e cria uma nova entrada nela.	Apaga o item da coleção.

1.5 Considerações finais do capítulo

A forma de projetar a arquitetura de sistemas passou por várias fases, acompanhando os avanços do poder computacional e da Internet. As aplicações corporativas evoluíram para acompanhar os desafios que lhes foram impostos: desde atender centenas de funcionários dentro de uma organização até atender milhões de clientes através da Internet.

A Arquitetura Orientada a Serviços apresentou uma maneira de compor sistemas diferente das arquiteturas divididas em *tiers* que eram usadas até então. Na SOA, um negócio é abstraído como um conjunto de serviços, que poderão ser implementados de diversas formas. Isso acabou permitindo que as arquiteturas se tornassem mais flexíveis.

A partir da década de 2010 um novo modelo arquitetônico – denominado como microserviços - ganhou força, principalmente em ambientes corporativos. Por ser o objeto principal de estudo deste projeto, ele será detalhado no Capítulo 2.

2 ARQUITETURA DE MICROSERVIÇOS

Segundo Newman (2015), a crescente necessidade das empresas de entregar software de forma rápida, empregando técnicas de automação de infraestrutura, testes e entrega contínua, fez com que os conceitos que fundamentam o design dos softwares fossem repensados. Se a arquitetura de um sistema não permite que mudanças sejam feitas de forma rápida, então haverá limites ao que pode ser entregue. Empresas começaram a fazer experimentos com arquiteturas de granularidade mais fina para atender estes e outros requisitos como melhor escalabilidade, maior autonomia de equipes e facilidade de adotar novas tecnologias.

Segundo Fowler (2014), o estilo arquitetural de microserviços pode ser definido como uma abordagem ao desenvolvimento de uma aplicação como um conjunto de pequenos serviços autônomos, cada um rodando em seu processo e comunicando-se através de mecanismos leves como uma API oferecida via HTTP. Estes serviços são construídos levando em consideração as capacidades do negócio, e seu *deploy* é feito de forma independente por um mecanismo completamente automatizado. Existe um mínimo de gerenciamento centralizado destes serviços, que podem ser escritos em linguagens diferentes e utilizar mecanismos de armazenamento de dados diferentes.

A arquitetura de microserviços pode ser considerada como um refinamento e simplificação da Arquitetura Orientada a Serviços (AMARAL, 2015).

2.1 Arquiteturas monolíticas

Segundo Fowler (2014), para compreender melhor a arquitetura de microserviços, é interessante compará-la à tradicional arquitetura monolítica. É comum ver aplicações empresariais construídas em três partes: uma interface para o usuário no lado do cliente – que consiste em páginas HTML e Javascript rodando no navegador da máquina do usuário, um banco de dados – que consiste em várias tabelas inseridas em um SGBD – e uma aplicação no lado do servidor. A aplicação do lado do servidor irá gerenciar requisições HTTP, executar a lógica de domínio, buscar e atualizar dados no banco de dados e popular páginas HTML para enviá-las ao navegador.

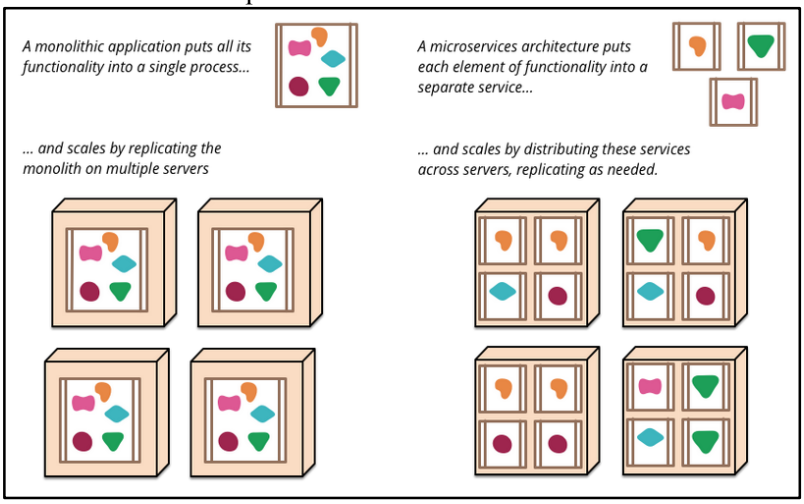
Esta aplicação no lado do servidor é considerada um monolito, pois trabalha como uma unidade única. Todas as alterações no sistema requerem a construção e o *deploy* de uma nova versão da aplicação. É um abordagem natural ao construir sistemas. Toda a lógica para tratar requisições está em um processo único, permitindo o aproveitamento de vários recursos de

linguagens de programação como a divisão em classes e funções. Com o devido cuidado, é possível até rodar e testar aplicações monolíticas na máquina do desenvolvedor. Ao utilizar um fluxo de *deploy* é possível garantir que mudanças serão testadas e disponibilizadas no ambiente de produção (FOWLER, 2014).

Aplicações monolíticas podem ter sucesso, mas um número crescente de equipes de desenvolvimento estão se frustrando com elas, especialmente quando um número crescente de aplicações precisa ser disponibilizado na nuvem. Mudanças são acopladas, fazendo com que alterações em uma pequena parte da aplicação exija um novo build e *deploy* do monolito inteiro. Escalar o monolito significa escalar a aplicação como um todo, ao invés de escalar somente as partes que requerem mais recursos (FOWLER, 2014).

Na Figura 7 é ilustrada e descrita a diferença entre escalar uma arquitetura monolítica e uma arquitetura baseada em microserviços.

Figura 7 – Escalando arquiteturas monolíticas e baseadas em microserviços



Fonte: Fowler, Microsevices, 2014

Aplicações monolíticas também se tornam difíceis de entender e modificar, especialmente quando elas começam a crescer e novos membros precisam ser adicionados ou substituídos nas equipes de desenvolvimento (NAMIOT, 2014).

A grande base de código presente nas aplicações monolíticas reduz a produtividade, diminui a qualidade do código, acaba com a modularidade e impede que desenvolvedores trabalhem de forma independente. Times inteiros precisam coordenar os esforços para desenvolvimento e *deploy* (NAMIOT, 2014).

Estas características acabaram levando à arquitetura de microserviços, uma tentativa de evitar as principais desvantagens de arquiteturas monolíticas.

2.2 Características chave dos microserviços

Os microserviços são pequenos e focados em realizar uma única tarefa. Eles são limitados de acordo com os limites do negócio. Isto faz com que se torne óbvio onde um trecho de código específico reside. Ao manter o foco nos limites do serviço, o crescimento desnecessário do serviço é evitado junto com todas as implicações que isto pode trazer (NEWMAN, 2015).

Eles também são autônomos. Cada microserviço é um entidade separada. Seu *deploy* pode ser feito como um serviço isolado em um PaaS ou como um processo em um sistema operacional. Toda a comunicação entre serviços é feita através de chamadas na rede utilizando APIs, para forçar sua separação. O gerenciamento centralizado destes microserviços, quando existente, também é um serviço completamente separado (NAMIOT, 2015).

2.3 Principais vantagens

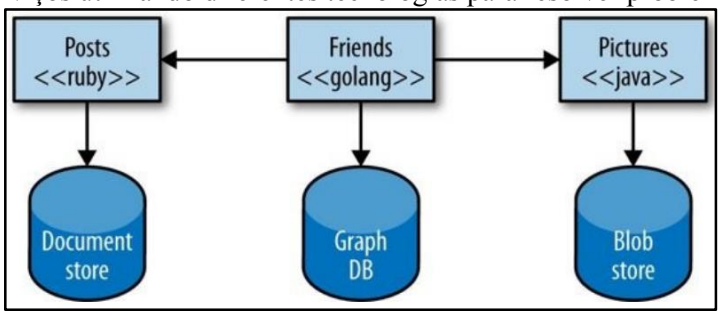
Segundo Newman (2015), os microserviços oferecem uma grande diversidade de vantagens, muitas delas inerentes à computação distribuída. Os microserviços conseguem tirar um maior proveito desses benefícios devido ao nível avançado a que eles levam os conceitos de sistemas distribuídos e da Arquitetura Orientada a Serviços.

2.3.1 Heterogeneidade tecnológica

Um dos benefícios chave da utilização de uma arquitetura baseada em microserviços é a heterogeneidade tecnológica. Com um sistema composto por múltiplos serviços que colaboram entre si, fica mais fácil usar diferentes tecnologias dentro deles. Isto permite que sejam escolhidas ferramentas que sejam mais adequadas para situações específicas, ao invés de ter que escolher uma abordagem mais padronizada e generalizada que pode acabar não garantindo o desempenho necessário (NEWMAN, 2015; AMARAL, 2015).

Se uma parte de um sistema necessita de uma melhoria de performance, é possível escolher uma pilha tecnológica que consegue atingir de forma mais eficiente o nível de performance desejado. Na Figura 8 é ilustrada uma arquitetura heterogênea, onde cada serviço utiliza linguagens e tecnologias para armazenamento de dados específicos para os problemas que eles resolvem.

Figura 8 – Serviços utilizando diferentes tecnologias para resolver problemas diferentes.



Fonte: Newman, Building Microsevices, 2015

Isto também torna mais rápida a adoção de novas tecnologias. Uma das grandes barreiras na adoção de novas tecnologias é o conjunto de riscos que isto representa. Numa aplicação monolítica, a adoção de uma nova linguagem de programação, banco de dados ou framework afeta uma grande porção do sistema. Em um sistema composto por vários serviços, existem diversos lugares onde novas tecnologias podem ser testadas sem causar grandes impactos no sistema como um todo (NIELSEN, 2015; NEWMAN, 2015).

Mas adotar diferentes tecnologias pode causar um excesso na utilização de recursos devido à quantidade de serviços que precisará ser executada de forma independente. Newman (2015) defende que deve-se evitar juntar múltiplos serviços em uma única máquina, mas esta preocupação torna-se desnecessária quando relacionada à aplicações relativamente pequenas.

Algumas empresas empregam limitações em relação à escolha de linguagens. Netflix e Twitter, por exemplo, utilizam a Java Virtual Machine como plataforma, já que eles possuem domínio sobre a confiabilidade e performance deste sistema. Apesar disso, nenhuma das duas empresas utiliza somente uma pilha de tecnologia para todas as tarefas (NEWMAN, 2015).

2.3.2 Resiliência

Outro benefício da utilização da arquitetura baseada em microserviços é a resiliência. Um conceito chave na engenharia de resiliência é o chamado *bulkhead*. Se um componente de um sistema falha, mas essa falha não se propaga, o problema pode ser isolado e o resto do sistema funciona normalmente. As fronteiras dos serviços acabam se tornando os *bulkheads*. Se um serviço monolítico falhar, a aplicação como um todo para. Um sistema monolítico pode ser executado em diversas máquinas para reduzir a chance de falha, mas ao se utilizar microserviços é possível construir sistemas que lidam com a falha total de serviços e que reduzem as funcionalidades gradativamente (NEWMAN, 2015).

Para garantir que os microserviços consigam tirar proveito da resiliência, é necessário

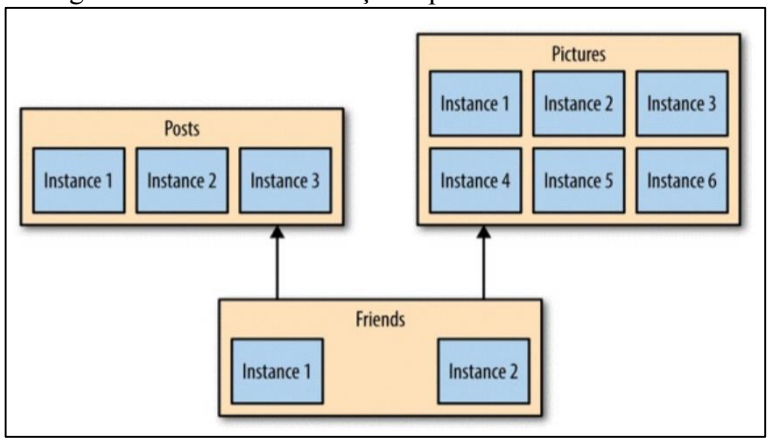
entender as fontes de falhas em sistemas distribuídos, como hardware*s* e redes. Se os arquitetos não souberem lidar com estas falhas e seus impactos, seus microserviços não serão resilientes.

2.3.3 Escalabilidade

A escalabilidade também é um benefício da arquitetura de microserviços. Ao utilizar uma grande aplicação monolítica, tudo precisa ser escalado junto. Se uma pequena parte de um sistema é o gargalo de performance, mas ela está contida numa aplicação monolítica, a aplicação como um todo precisará ser escalada (FOWLER, 2014; NAMIOT, 2014).

Com serviços menores, é possível escalar somente o que for necessário, permitindo rodar partes menores do sistema em hardware*s* mais modestos (AMARAL, 2015). Na Figura 9 é ilustrado como diferentes serviços podem ser escalados de forma separada.

Figura 9 – Diferentes serviços replicados individualmente



Fonte: Newman, Building Microsevices, 2015

Ao utilizar sistemas de provisionamento sob demanda como o Amazon Web Services, é possível aplicar a escala sob demanda somente para as partes que precisam. Isso permite que haja um controle mais efetivo sobre os custos (AMARAL, 2015).

2.3.4 Facilidade de *deploy*

A mudança de uma linha de código em um sistema monolítico que tem milhões de linhas significa que um *deploy* novo de todo o sistema deverá ser feito. Isto pode causar grandes impactos e envolver vários riscos. Na prática, este tipo de *deploy* acaba se tornando pouco frequente, fazendo com que muitas mudanças se acumulem entre lançamentos para a versão de produção. E quanto maior for o número de mudanças, maiores são os riscos envolvidos de

acontecer falhas (NEWMAN, 2015; NIELSEN, 2015).

Ao utilizar microserviços, mudanças podem ser feitas em um único serviço e lançadas de forma independente do restante do sistema. Isto permite que o *deploy* seja mais rápido. Se um problema ocorrer, ele pode ser isolado rapidamente a um serviço específico, facilitando a possibilidade de realizar *rollbacks*, por exemplo. Isto também significa que novas funcionalidades podem ser disponibilizadas para clientes de forma mais rápida. Esta é uma das razões pela qual empresas como Amazon e Netflix usam estas arquiteturas: garantir o mínimo de impedimentos ao lançar novas funcionalidades (NEWMAN, 2015).

A arquitetura de microserviços também apresenta desvantagens. Um dos fatores a ser levados em consideração é que ela carrega todas as complexidades relacionadas à sistemas distribuídos. Outro fator é que ao adotar este tipo de arquitetura, é necessário dominar técnicas para efetuar *deploy*, testes e monitoramento para poder usufruir das vantagens deste modelo.

Desta forma, é preciso lembrar que cada empresa, organização e sistema é diferente, fazendo com que determinadas soluções possam se encaixar em determinados cenários, mas não em outros. Antes de adotar uma arquitetura baseada em microserviços, é necessário analisar todos os fatores envolvidos, para determinar se os microserviços realmente devem ser adotados e até que nível.

2.4 Identificação e modelagem de microserviços

Segundo Newman (2015), a atividade de modelagem de serviços requere foco em dois conceitos principais, que são comuns no contexto de sistemas orientados a objetos: baixo acoplamento e alta coesão.

Alta coesão: é desejável que comportamentos semelhantes estejam agrupados e comportamentos diferentes estejam separados. Isto permite que alterações em determinados comportamentos sejam feitas em um único lugar e lançadas o mais rápido possível.

Se mudanças em comportamentos tiverem que ser feitas em lugares diferentes, vários serviços precisarão ser lançados (talvez ao mesmo tempo). Realizar alterações em diversos lugares é mais demorado e fazer o *deploy* de vários serviços ao mesmo tempo é arriscado – dois cenários que devem ser evitados.

Baixo acoplamento: quando serviços possuem baixo acoplamento, uma mudança interna em um serviço não deve causar implicações em outro serviço.

A vantagem principal na utilização de microserviços é a possibilidade de alterá-los e fazer seu *deploy* sem a necessidade de modificar nenhuma outra parte do sistema.

Um serviço com baixo acoplamento conhece somente o estritamente necessário sobre os outros serviços com os quais ele colabora. Isso significa que é desejável limitar as diferentes chamadas que um serviço faz a outro, porque além de um possível problema de performance, a comunicação verbosa pode levar ao alto acoplamento.

Um dos aspectos que pode influenciar negativamente o acoplamento de serviços é escolher um estilo de integração que liga fortemente serviços, fazendo com que mudanças em um serviço também cause mudanças em outros serviços.

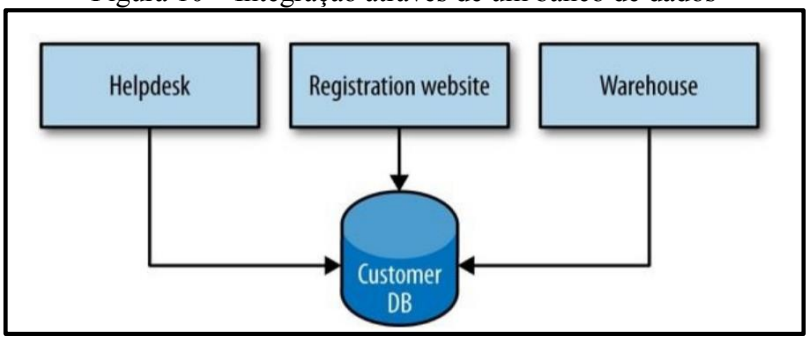
2.5 Aspectos de integração e comunicação

Segundo Newman (2015), fazer uma integração de forma correta é o aspecto mais importante ao se pensar em uma arquitetura baseada em microserviços. Às vezes é necessário fazer mudanças em serviços que forçam mudanças em seus consumidores. A escolha de tecnologia deve ser feita de forma que garanta que isto ocorra o mais raramente possível.

2.5.1 Tecnologias para integração

Segundo Newman (2015), uma das formas mais comuns de integração presentes na indústria de software é através de bancos de dados. Ao utilizar esta abordagem, aplicações acessam o banco de dados diretamente para realizar operações CRUD. É um tipo de integração simples e rápida de implantar, fatores que contribuem para sua popularidade. Na Figura 10 são ilustrados três serviços diferentes acessando diretamente o banco de dados.

Figura 10 – Integração através de um banco de dados



Fonte: Building Microservices, 2015

Apesar de muito utilizado, este modelo apresenta problemas. Um deles é que ao

permitir que agentes externos visualizem e façam seus modelos baseados em detalhes de implementação internos, o alto acoplamento está sendo promovido. Isso acaba fazendo com que eventuais mudanças na estrutura do banco de dados quebrem a compatibilidade dos clientes. Desta forma, este processo de mudança pode acabar se tornando complexo.

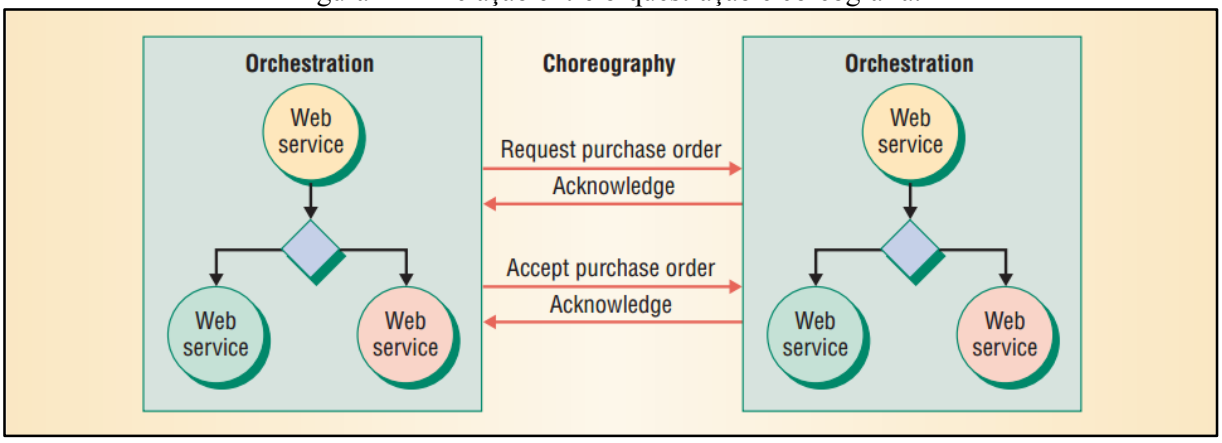
Outro problema é a limitação na escolha de tecnologias. Ao integrar diretamente com um banco, os consumidores estão presos à tecnologia de um banco específico. Se uma empresa decidir migrar seu banco de dados relacional para um banco NoSQL, por exemplo, todos os clientes que foram integrados diretamente com o banco serão afetados.

A integração entre serviços deve ser feita de forma que suas APIs de comunicação sejam agnósticas em relação à tecnologias. Isso significa evitar tecnologias de integração que ditam a pilha tecnológica que é utilizada para construí-los. Ela também devem ser feita de forma desacoplada, evitando que alterações internas em suas estrutura não afete os clientes que os consomem (NEWMAN, 2015).

2.5.2 Orquestração e coreografia

Segundo Peltz (2003), os termos orquestração e coreografia descrevem dois aspectos da criação de processos de negócios a partir de *webservices* compostos. Os dois termos acabam se sobrepondo de certa forma. Na Figura 11 é ilustrada a sua relação em alto nível.

Figura 11 – Relação entre orquestração e coreografia.



Fonte: Peltz, Web Services orchestration and choreography, 2003

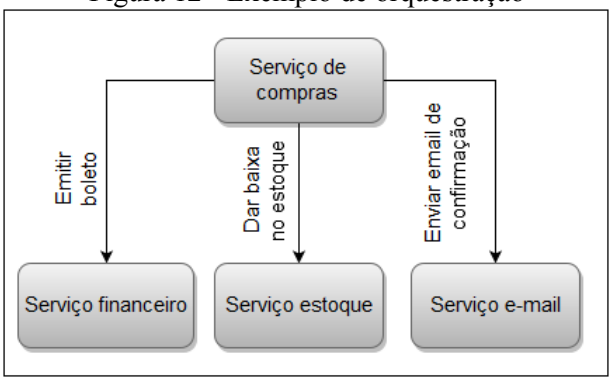
Orquestração se refere a um processo de negócio executável que pode se relacionar com *webservices* internos e externos. As interações ocorrem em nível de mensagem. Elas incluem regras de negócio e ordem de execução de tarefas, podendo se estender à aplicações e organizações para definir um modelo de processo com tempo de vida longo, transacional de

com várias etapas (Peltz, 2003).

A orquestração sempre representa o controle da perspectiva de uma das partes. É diferente da coreografia, que é mais colaborativa e permite que cada parte envolvida descreva seu papel na integração. A coreografia rastreia mensagens entre diversas partes e fontes, normalmente as mensagens públicas trocadas entre *webservices,* em vez de rastrear um processo de negócio específico que uma única parte executa (PELTZ, 2003).

Na Figura 12 é apresentado um exemplo mais concreto de orquestração, onde um serviço responsável pela consolidação de compras age como uma peça central. Após a consolidação de uma compra, ele é responsável por notificar individualmente o serviço financeiro para a emissão de um boleto, o serviço de estoque para a baixa dos produtos e o serviço de e-mail para o envio da confirmação. É possível garantir o sucesso de todas as etapas ao utilizar comunicação síncrona, sempre aguardando a resposta de cada serviço.

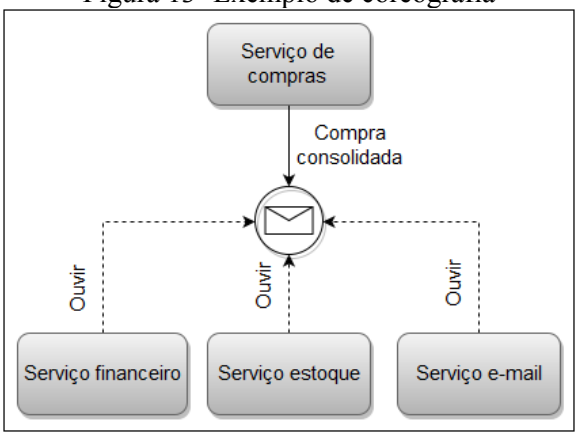
Figura 12 - Exemplo de orquestração



Fonte: Elaborado pelo autor, 2015

O mesmo cenário pode utilizar a coreografia para resolver o mesmo problema, como ilustrado na Figura 13. Os serviços financeiro, de estoque e de e-mail ficariam aguardando por um *broadcast* do evento de consolidação de uma compra e realizariam suas funções de acordo com esta informação.

Figura 13- Exemplo de coreografia



Fonte: Elaborado pelo autor, 2015

Desta forma, o serviço de compra tem menos responsabilidades e se torna mais desacoplado. Também ficaria mais fácil implantar uma comunicação assíncrona entre os serviços, já que o serviço de compras não precisaria aguardar pela resposta dos outros serviços. A desvantagem desta abordagem é que seria preciso criar mecanismos para garantir que os outros serviços receberam o *broadcast* e realizaram suas respectivas funções.

2.6 Exemplos de casos

Diversas empresas estão migrando suas aplicações para o conceito de microserviços. O texto que segue descreve casos de sucesso na utilização deste tipo de arquitetura em cenários reais.

2.6.1 Nevada Research Data Center

O Solar Nexus Project é um esforço colaborativo entre cientistas, engenheiros educadores e técnicos no estado de Nevada, nos Estados Unidos, que visa aumentar a utilização de energia solar renovável e reduzir seus efeitos adversos no ambiente e em animais selvagens, ao reduzir o consumo de água. O projeto tem como objetivo pesquisar diversas áreas, incluindo o uso de água nas usinas elétricas, os efeitos da construção de novas usinas no ambiente, utilização de água de outras fontes para manter as usinas e soluções interdisciplinares para melhorar a geração de energia solar em Nevada (VINH, 2015).

Para organizar e analisar dados que pudessem produzir mudanças efetivas, o Nexus precisava de um banco de dados centralizado para armazenar dados coletados. Com este intuito foi construído o Nevada Climate Change Portal (NCCP), que depois foi substituído pelo Nevada

Research Data Center, ou NRDC (VINH, 2015).

Em 2013, o NCCP foi criado para armazenar dados iniciais do projeto. Com o passar do tempo, o projeto Nexus evoluiu, mas sua base de dados não. O NRDC foi criado para substituir o NCCP e corrigir diversos problemas que o primeira tinha, como sua flexibilidade, seu escopo e sua complexidade.

Mas o NRDC acabou herdando diversas falhas da arquitetura monolítica de seu predecessor. Não existiam formas de monitoramento de rede, a manutenção do sistema exigia a suspensão completa do serviço e a escalabilidade era limitada. Um time foi designado para reconstruir o sistema utilizando uma arquitetura baseada em microserviços, uma melhoria em relação à arquitetura monolítica original. Esta nova arquitetura permitiu que o NRDC tivesse um *deploy* de serviços mais rápido, recursos escaláveis e armazenamento confiável de dados (VINH, 2015).

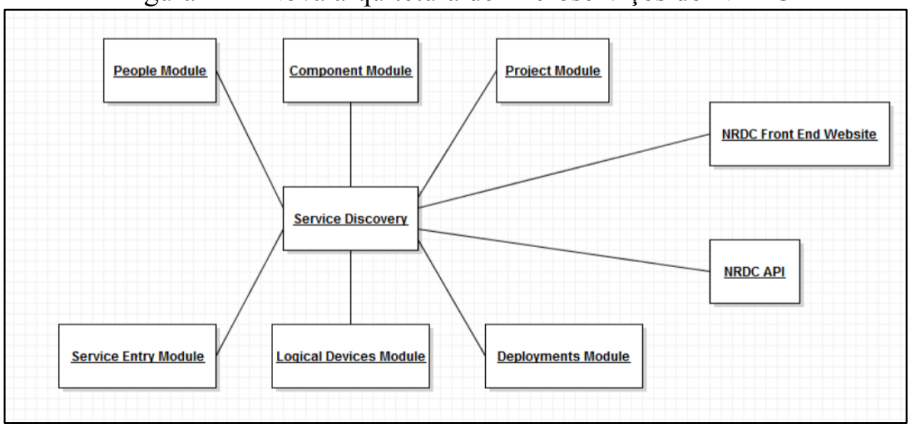
O protótipo redesenhado do NRDC visa atacar cada um dos problemas mencionados e prover um sistema robusto e eficiente. Em vez de continuar com a arquitetura antiga, o propósito do redesign é refazer a arquitetura, com microserviços em mente. Isto significa que serão criados diversos serviços encapsulados que existem fora do escopo dos outros, completamente autônomos e que desconhecem a existência de outros serviços. Cada serviço será independente e utilizará um serviço de registro para fazer sua comunicação. A nova arquitetura trouxe uma grande mudança em duas formas. A primeira é que o novo NRDC permanece sempre ativo, mesmo durante manutenções. Desligar um serviço não afetará o site como um todo, já que outros serviços estarão vivos para suportar o site. A segunda consiste na possibilidade de serviços serem trocados, adicionados ou removidos. Esta característica permitirá uma escalabilidade maior, garantindo um enorme potencial de crescimento (VINH, 2015).

Apesar do protótipo do NRDC ter sido baseado somente no projeto Nexus Track 1, a ideia de microserviços não foi. Durante o desenvolvimento do projeto, modelos distribuídos como o da aplicação de streaming Netflix e a Amazon.com serviram como principais fontes de inspiração. Tanto o Netflix como a Amazon usam arquiteturas de microserviços na nuvem, hospedadas na Amazon Web Services. A principal diferença entre o site NRDC e a Amazon ou o Netflix será que seus microserviços rodarão em servidores físicos em um primeiro momento, ao invés de rodarem em instâncias na cloud AWS (VINH, 2015).

A princípio foram definidos seis serviços básicos que irão compor os requisitos básicos da arquitetura: Pessoas, Sistema, Entradas de Serviço, *Deploys* e Projetos. Apesar de compartilharem diversos relacionamentos, os serviços se manterão independentes e se

comunicarão por meio de uma Descoberta de Serviços. Também foi estabelecido como requisito básico a habilidade de monitorar o tráfego da rede, detectar e gerenciar falhas e coletar estatísticas como acessos ao website. Na Figura 14 é ilustrada a distribuição da arquitetura na primeira fase do projeto.

Figura 14 – Nova arquitetura de microserviços do NRDC



Fonte: Vinh D. Le, et al., Microservice-based Architecture for the NRDC, 2015

O redesign do NRDC consiste em cinco grandes partes: módulos, Descoberta de Serviços, website, banco de dados e uma API. Os módulos substituíram a típica aplicação monolítica que reside no lado do servidor, e utilizam a descoberta de serviços como um tipo de DNS local para saber a localização de outros serviços. Eles são diversos serviços que aceitam entradas vindas somente da Descoberta de Serviços para buscar e apresentar informações vindas do banco de dados. A Descoberta de Serviços age como uma tabela de busca para encontrar a localização de outros módulos por meio do seu endereço IP e porta (VINH, 2015).

O NRDC utilizou um conjunto de diferentes tecnologias para implementar sua arquitetura. O Eureka da Netflix foi utilizado como sistema de Descoberta de Serviços. Ele permite a função de descoberta e monitoramento de serviços. O Eureka foi projetado inicialmente para trabalhar em conjunto com a AWS em sistemas distribuídos na nuvem.

Os módulos foram desenvolvidos utilizando a linguagem Python, escolhida pela facilidade de uso, pela variedade de bibliotecas que oferece e pela facilidade de aprendizado, tento em vista as futuras equipes que farão a manutenção dos módulos. O Flask, microframework web para Python, foi escolhido por permitir o *deploy* de módulos como servidores, que podem ser reaproveitados para módulos futuros. Como tecnologia para armazenamento de dados foi utilizado o PostgreSQL (VINH, 2015).

A arquitetura proposta trouxe benefícios significativos para o projeto NRDC, como a escalabilidade, confiabilidade e mantenabilidade. Já existem projetos futuros pra migrar toda a

infraestrutura para a nuvem, alocando cada serviço em uma instância e otimizando o uso de hardware e eficiência. Migrar para cloud eliminaria restrições de hardware e permitiria o acesso infinito a recursos, que poderiam ser escalados para cima ou para baixo em paralelo com os serviços. Também existe a possibilidade de implementar soluções de Big Data como o Apache Hadoop ou Google MapReduce para facilitar a análise do grande volume de dados existente no sistema (VINH, 2015).

2.6.2 Netflix

A Netflix nasceu no ano de 1998 como uma empresa de aluguel de filmes em mídia física. Clientes poderiam alugar filmes e estes seriam enviados via correio para seu endereço. Nos anos seguintes a empresa se popularizou devido ao seu modelo diferenciado de negócio, que contava com aluguel ilimitado, sem taxas de atraso, devolução ou entrega. A partir do ano de 2008, a Netflix passou a oferecer para seus assinantes a opção de assistir filmes através de *streaming* na Internet.

Nos anos de 2009 e 2010 a Netflix começou a migração de seu sistema para cloud. Essa escolha foi feita devido ao tempo necessário e aos grandes custos envolvidos na construção de datacenters dedicados. A Amazon Web Services foi a empresa escolhida para abrigar sua infraestrutura. Ela faz a cobrança no mês seguinte ao uso, permitindo que os investimentos do Netflix sejam focados em adquirir novo conteúdo e as despesas sejam feitas para entregar este conteúdo (COCKCROFT, 2015).

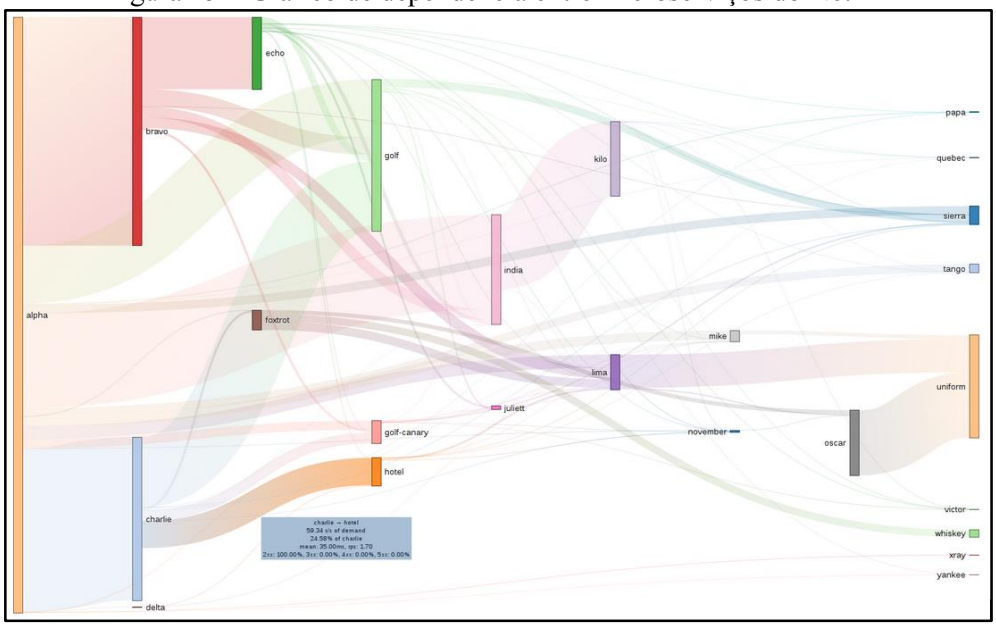
A transição para a cloud aconteceu de forma gradativa. A primeira parte da Netflix a rodar na nuvem foi um serviço de *auto-complete*. Quando as pessoas começam a digitar parte de uma palavra, o site sugere opções baseadas no que foi digitado. Quando este serviço foi implementado, todo o site ainda rodava em um datacenter dedicado. Foi uma tecnologia trivial, mas de grande importância para a equipe, pois os ensinou a subir um sistema para a nuvem, conectá-lo a um balanceador de carga e a utilizar todas as ferramentas necessárias para isso. Este projeto, apesar de pequeno, demorou um mês para ser implantado com sucesso (COCKCROFT, 2015).

A partir daí, a empresa começou a decompor seu sistema em diversos serviços. A arquitetura evoluiu de um único WebApp (.war) em 2008 para centenas de pequenos serviços em 2012. A princípio, a Netflix não denominava sua nova arquitetura como microserviços. Eram utilizados termos como “nativo na cloud” ou “SOA de granularidade fina”. O termo microserviços acabou sendo adotado pela Netflix após ser sugerido a eles pela equipe da

ThoughtWorks (COCKCROFT, 2015).

O gráfico de dependência da Figura 15 ilustra a relação entre alguns dos microserviços da Netflix. Nele é possível ver alguns dos serviços utilizados pela Netflix, a forma como se comunicam entre si e quanto tempo eles gastam na comunicação.

Figura 15 – Gráfico de dependência entre microserviços do Netflix



Fonte: Netflix, A Microscope on Microservices, 2015.

A Netflix se tornou pioneira em novas arquiteturas e tecnologias cloud para operar em escala massiva – uma escala que leva ao limite diversos tipos de tecnologias. Isto faz com que eles tenham que desenvolver várias ferramentas próprias. Grande parte destas tecnologias são abertas ao público por meio do Netflix OSS (Netflix Open Source Software Center). O desafio não é só oferecer funcionalidades e gerenciar seu número massivo de instâncias, mas também prover *insights* rápidos e que possam gerar ações para uma arquitetura baseada em microserviços de grande escala.

2.7 Considerações finais do capítulo

A arquitetura baseada em microserviços herdou diversas características da Arquitetura Orientada a Serviços, e acabou levando seus conceitos a outro nível. Diferente do SOA, a arquitetura de microserviços é baseadas em serviços de granularidade fina e com mensagens pequenas. O surgimento dos microserviços está diretamente relacionado com a crescente necessidade de migração para a nuvem, com a necessidade de entregar software de forma mais ágil e garantir que o software entregue seja resiliente.

Mas adotar a arquitetura de microserviços traz diversas complicações. Para que seja

implantada com sucesso, é necessário adotar uma nova cultura, mudando a forma de pensar no projeto, no desenvolvimento e na entrega do software.

3 UMA ARQUITETURA BASEADA EM MICROSERVIÇOS

O projeto implantado tem como intuito demonstrar as vantagens e desvantagens da utilização de microserviços durante o ciclo de desenvolvimento de aplicações corporativas. Por esta razão, foi idealizado um projeto que visa atender as demandas de um cenário real, envolvendo a utilização de um aplicativo móvel.

3.1 Projeto

A aplicação idealizada permite que usuários efetuem buscas por diversos tipos de produtos e visualizem seus preços em lojas físicas próximas, por meio da utilização de um *smartphone*. Para isso, a aplicação conta com um registro de preços de produtos em diferentes estabelecimentos. Os usuários tem a opção de cadastrar preços para garantir que estes se mantenham atualizados. A aplicação também permite que o usuário sinalize interesse em determinados produtos para receber notificações sobre alterações de preço e promoções em estabelecimentos próximos e informações relevantes sobre suas preferências.

A aplicação está dividida em duas partes principais: um aplicativo móvel responsável por oferecer uma interface com os usuários e coletar dados através dela e um *backend* responsável por manter os dados e oferecê-los de forma inteligente para o aplicativo móvel.

A aplicativo móvel e o *backend* foram desenvolvidos de forma desacoplada. Isso significa que as duas partes não conhecem os detalhes internos de implementação do outro, somente suas interfaces de comunicação. Toda a comunicação é feita por meio desta interface padronizada, permitindo que o *backend* e o aplicativo móvel evoluam até certo ponto de forma independente.

3.2 Funcionalidades da aplicação

As funcionalidades implementadas na aplicação podem ser encontradas em diversos sistemas modernos, com adaptações para cenários específicos. Grande parte delas serão implementadas no *backend* e disponibilizadas para consumo do aplicativo móvel.

3.2.1 Busca georreferenciada de preços

O aplicativo móvel permite que usuários pesquisem preços de diversos tipos de

produtos em lojas físicas próximas aos usuários. O fluxo da busca é dividido em duas etapas.

Na primeira etapa o usuário busca por um produto. Ele tem a opção de utilizar o código de barras do produto ou palavras-chave como marca, fabricante e categoria para efetuar a busca.

Na segunda etapa, são buscados preços do produto em estabelecimentos próximos ao usuário. Para oferecer uma experiência personalizada para o usuário e economizar recursos do dispositivo móvel e do *backend*, os preços exibidos nesta etapa são limitados a um raio de distância fixo em relação à posição do usuário.

3.2.2 Cadastro de produtos

Se o usuário efetuar uma busca e esta não retornar nenhum resultado, ele tem a opção de cadastrar um novo produto. Para garantir que os dados inseridos sejam confiáveis, o aplicativo contará com um mecanismo que faz uma busca na Internet, obtendo informações do produto a partir de seu código de barras. Se forem encontradas informações sobre o produto, elas serão automaticamente preenchidas na tela de cadastro.

3.2.3 Cadastro de preços

Ao constatar que a informação sobre o preço de um produto em um estabelecimento está desatualizada, errada ou não existe, o usuário poderá cadastrar um novo preço referente a um estabelecimento, sinalizando também se este é promocional ou não.

3.2.4 Lista de interesses

O aplicativo permite que usuários marquem produtos como “favoritos”, criando assim uma lista de interesses. Desta forma o usuário consegue facilmente acompanhar a variação de preço dos produtos em sua lista. Ele também receberá *push notifications* sempre que produtos da sua lista de interesse tiverem preços promocionais adicionados em estabelecimentos próximos.

3.2.5 Lista de promoções

O aplicativo conta com uma lista de promoções. Nesta lista são exibidos produtos que tiveram preços cadastrados como promocionais. Assim como a busca de produtos, a lista de

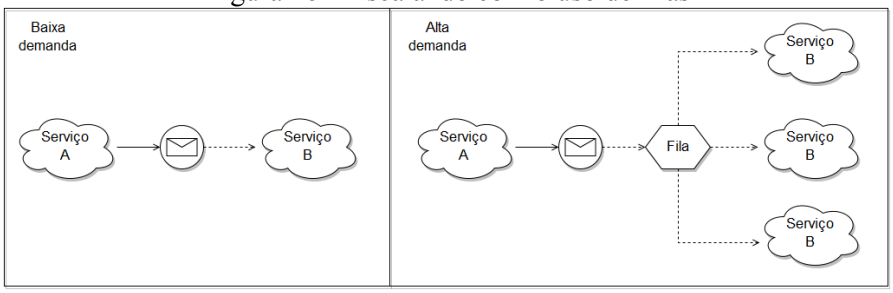
promoções também é georreferenciada. As promoções exibidas são limitadas a um raio de distância pré-determinado em relação ao usuário.

3.3 Arquitetura geral do *backend*

O *backend* é responsável por armazenar, processar e disponibilizar as informações que são consumidas pelo aplicativo móvel. Suas funcionalidades finais correspondem às funcionalidades disponibilizadas no aplicativo móvel.

A comunicação entre os microserviços acontece por meio da coreografia assíncrona. Serviços que necessitam de informações de outros serviços deverão fazer uma inscrição para recebê-las. A escolha dessa abordagem garante um melhor desempenho dos serviços transmissores, permitindo que estes não fiquem bloqueados ao enviar mensagens. Este mecanismo também garante uma maior escalabilidade, permitindo que filas sejam implementadas. Isto facilita a replicação de serviços ouvintes em cenários de alta demanda, como ilustrado na Figura 16.

Figura 16 – Escalando com o uso de filas

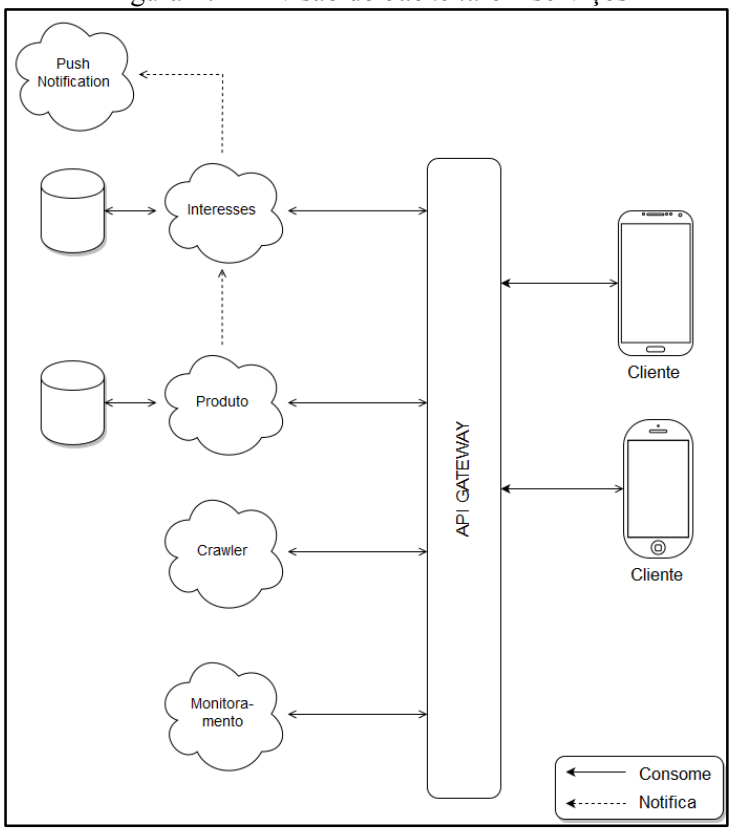


Fonte: Elaborado pelo autor, 2015

A desvantagem da utilização da coreografia assíncrona neste projeto é a falta de um mecanismo que garante a entrega das mensagens. Os serviços transmissores não dão nenhuma garantia de que as mensagens foram entregues para os ouvintes, e também não armazenam as mensagens caso não existam ouvintes. Neste projeto, a experiência oferecida ao usuário pelo mecanismo de *push notifications* seria diretamente afetada por esta deficiência.

Na Figura 17 é ilustrada a divisão dos serviços implementados no *backend*.

Figura 17 – Divisão do *backend* em serviços



Fonte: Elaborado pelo autor, 2015

A seguir será descrita a divisão das responsabilidades em serviços abstratos. Esta divisão é feita para definir "barreiras" entre partes do sistema. Cada serviço abstrato poderá ser implementado na forma de um ou mais microserviços.

3.3.1 Serviço de produtos

O serviço de produtos é responsável por manter informações sobre produtos e seus preços em estabelecimentos e disponibilizá-las de forma inteligente para o consumo por outros serviços ou aplicações através de uma interface uniforme. Ele está diretamente relacionado com a maior parte das informações consumidas pelo aplicativo móvel, como a consulta e cadastro de produtos e preços.

A vantagem na utilização deste serviço é a possibilidade de evoluir os modelos e as funcionalidades oferecidos por ele, sem afetar outros serviços.

3.3.2 Serviço de interesses

O serviço de interesses armazena todas as listas de interesses dos usuários, suas

localizações aproximadas e os tokens para o envio de notificações. Ele se inscreverá para receber notificações de atualizações de preços vindas do serviço de produtos. O aplicativo móvel não irá consumir diretamente informações deste serviço, somente alimentá-lo com registros de interesses.

A vantagem na utilização deste serviço é a possibilidade de implementar diferentes regras para o envio de *push notifications* de forma independente dos outros serviços. A primeira regra a ser implementada será o envio de notificações sempre que um novo preço promocional for cadastrado em um estabelecimento que esteja em um raio próximo ao usuário, e este tenha sinalizado interesse no produto.

3.3.3 Serviço de push notifications

A função do serviço de *push notifications* é reencaminhar uma notificação para os serviços de notificações das principais plataformas mobile, como o Google Cloud Messaging, o Apple Push Notification Server e o Microsoft Push Notification Service.

Para realizar suas funções corretamente, um modelo comum de notificação que se aproxima do modelo específico de cada plataforma foi definido. Assim é possível manter uma padronização nos dados de entrada independentemente da plataforma alvo da notificação.

A vantagem da criação deste serviço é a flexibilidade que ele oferece referente ao envio de notificações. Outros serviços que queiram enviar *push notifications* não precisarão se adequar ao funcionamento específico de cada plataforma. Este serviço se encarregará de traduzir automaticamente as notificações para atender às interfaces dos serviços externos.

3.3.4 Serviço *crawler*

O serviço *crawler* é responsável por coletar informações na Internet sobre produtos que ainda não foram cadastrados no microserviço de produtos. Ele será acionado quando um usuário estiver cadastrando um novo produto no aplicativo móvel e informar um código de barras. Se informações forem encontradas, elas serão apresentadas ao usuário, evitando que ele precise digitar muitos dados.

A vantagem da criação deste serviço separado do serviço de produtos é a possibilidade de evoluir sua funcionalidade de forma independente. Diversas fontes diferentes de informação poderão ser adicionadas para consulta, tendo seus dados normalizados para atender a um formato comum. Tudo isso sem a necessidade de alteração em outros serviços.

3.3.5 Serviço de monitoramento e estatísticas

O serviço de monitoramento é responsável por coletar e disponibilizar diferentes tipos de dados gerados pelos serviços como *endpoints* consultados, termos pesquisados, tempo de resposta, erros de aplicação, entre outros. Todos os dados gerados pelos serviços serão disponibilizados para consulta em tempo real, permitindo a reação imediata a falhas e também a consulta de dados estratégicos.

3.3.6 Gateway de APIs

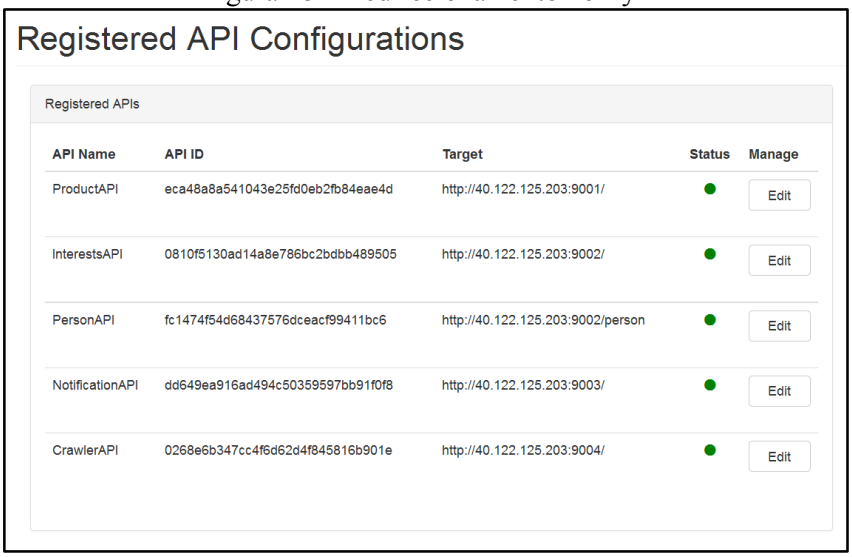
De acordo com a divisão estabelecida, o aplicativo móvel pode se comunicar diretamente com três serviços diferentes. Estes serviços rodam de forma independente, o que significa que eles podem responder em portas ou endereços completamente diferentes.

Para abstrair a divisão interna da arquitetura e prover autenticação para consumidores externos, será utilizado o Tyk, um gateway e plataforma de gerenciamento de APIs open source que permite fazer um controle fino de acesso, gravar estatísticas detalhadas de acesso de usuários e de erros. Como um gateway de APIs, ele fica na frente de aplicações sendo responsável por fazer a autorização, controle de acesso e limitação do *throughput* de serviços. Ele permite que o foco do desenvolvedor esteja na implementação correta dos serviços, em vez da infraestrutura (TYK, 2015).

Apesar de ter o foco em proteger APIs de acesso não autorizado, o Tyk oferece diversas funcionalidades como balanceamento de carga, endpoints virtuais, cache, monitoramento ativo, transformações de requisições e respostas, versionamento de APIs, políticas de acesso, dentre outras (TYK, 2015).

A utilização do Tyk permite que seja definido um endereço e portas únicos para que o aplicativo móvel consuma. Ele será responsável por redirecionar as requisições para o endereço adequado. Ele é utilizado de forma similar a um servidor DNS, com a vantagem de também permitir a autenticação dos usuários. Na Figura 18 é possível visualizar a divisão dos microserviços e seus endereços.

Figura 18 – Redirecionamento no Tyk



Fonte: Elaborado pelo autor, 2015

3.4 Principais tecnologias utilizadas

Utilizar uma pilha tecnológica adequada é fundamental para tirar proveito dos benefícios que a arquitetura de microserviços oferece. A seguir serão descritas as principais tecnologias utilizadas para implementar as funcionalidades dos serviços propostos.

3.4.1 Play Framework

O Play Framework é um framework *open source* de alta produtividade e que integra diversos componentes e APIs para o desenvolvimento de aplicações web modernas. Ele é baseado em uma arquitetura leve, sem estado e voltada para a web e conta com um consumo mínimo e previsível de processamento, memória e threads para aplicações altamente escaláveis graças ao seu modelo reativo, baseado no Iteratee IO. O Play Framework permite que aplicações sejam desenvolvidas utilizando as linguagens Java e Scala. (PLAY FRAMEWORK, 2015).

Aplicações desenvolvidas com Play Framework rodam de forma independente. Ao fazer seu *deploy*, um arquivo JAR é gerado contendo a aplicação, todos seus arquivos e um servidor web (Netty) embutido. O único requisito para executar uma aplicação Play é a presença da JVM. (PLAY FRAMEWORK, 2015).

3.4.2 Scala

A linguagem Scala foi utilizada em conjunto com o Play Framework na construção de alguns dos microserviços. Ela reúne características da programação orientada a objetos e da programação funcional. O código escrito em Scala é compilado para rodar na JVM, tirando proveito da sua maturidade e estabilidade (SCALA, 2015).

A linguagem Scala permite interoperabilidade com o Java. Isto significa que pilhas Java e Scala podem trabalhar em conjunto de forma transparente. Classes escritas em Scala podem chamar métodos Java, criar objetos Java e implementar interfaces Java (SCALA, 2015).

Ela foi escolhida por possuir uma sintaxe concisa e simples de entender, permitindo implementar funcionalidades tão complexas quanto o Java, mas gerando menos linhas de código.

3.4.3 Slick

Alguns microserviços contam com um banco de dados próprio. A camada de acesso ao banco destes microserviços utilizará o Slick, um mapeador funcional-relacional (FRM) para Scala. O Slick permite que o acesso ao banco seja feito utilizando uma sintaxe semelhante à de acesso a coleções ou SQL direto quando necessário. Todas as ações do banco são feitas de forma assíncrona, tornando-o uma boa escolha para ser utilizado em conjunto com o Play Framework (SLICK, 2015).

O Slick não é um ORM como o Hibernate, apesar dos dois serem soluções para persistência de dados e compartilharem alguns conceitos. Uma das principais diferenças entre o Slick e alguns ORMs tradicionais é a configuração necessária para acessar o banco. Alguns ORMs requerem diversos arquivos de configuração, enquanto o Slick precisa de pequenos trechos de código Scala para isso (SLICK, 2015).

Neste projeto o Slick é utilizado para manter a independência do SGBD utilizado pelos microserviços, já que ele se encarrega de traduzir comandos Scala em comandos SQL para diferentes SGBDs de forma transparente para o desenvolvedor.

3.4.4 Java

A linguagem Java também foi utilizada em conjunto com o Play Framework na construção de alguns microserviços. Ela é uma linguagem de propósito geral, orientada a

objetos e baseada em classes. O código gerado em Java é compilado para bytecode e executado na JVM (GOSLING, 2105).

Ela foi escolhida para implementação de alguns microserviços por apresentar uma curva de aprendizado baixa, permitindo que a manutenção de códigos pequenos seja simples.

3.4.5 PostgreSQL

A princípio, alguns dos microserviços foram projetados para utilizarem um armazenamento de dados relacional. Para este fim, será utilizado o PostgreSQL, um poderoso SGBD de código aberto. Ele conta com mais de 15 anos de desenvolvimento ativo e uma arquitetura que ganhou uma forte reputação graças à sua confiabilidade, integridade de dados e precisão (POSTGRESQL, 2015).

Apesar da escolha de utilizar um banco de dados relacional em alguns microserviços, possíveis migrações futuras para outros paradigmas de armazenamento como o NoSQL, por exemplo, não afetarão outros serviços, já que o acesso aos dados disponibilizados pelos microserviços é feito somente através de suas APIs REST.

3.4.6 Elasticsearch

O Elasticsearch é um dos microserviços que compõem a arquitetura. Ele é uma poderosa ferramenta de busca textual construída em volta do Apache Lucene. O Elasticsearch possui uma arquitetura altamente distribuída e oferece acesso às suas funcionalidades através de uma interface REST. Ele roda de forma independente e permite armazenar, indexar e disponibilizar para busca documentos em formato JSON (ELASTIC, 2015).

A vantagem em sua utilização está na possibilidade de tirar proveito de seu mecanismo de busca textual, sem a necessidade de implementar este mecanismo internamente. A desvantagem é o *overhead* de recursos, já que ele também se comportará como um processo independente, consumindo memória, processamento e disco. Isto pode ser mitigado pela possibilidade de alocar o Elasticsearch em uma máquina dedicada.

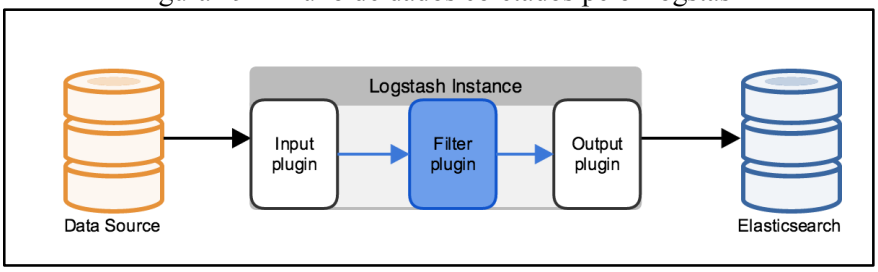
3.4.7 Logstash

O Logstash também é um microserviço que compõe a arquitetura. Ele é um canal de dados que auxilia no processamento de logs e outros dados de eventos gerados por diversos

sistemas. Ele consegue se conectar a uma variedade de fontes, normalizar esquemas variados de dados e transmiti-los para um sistema central de análises. A sua utilização em conjunto com o Filebeat permite que dados em diversas máquinas diferentes sejam coletados e centralizados (LOGSTASH, 2015).

Ele será utilizado para coletar os *logs* gerados pelos microserviços. Após recolhidos e normalizados, os *logs* serão enviados para o Elasticsearch para serem indexados e disponibilizados para a consulta. O Logstash se comunica de forma nativa com o Elasticsearch, dispensando configurações adicionais. Na Figura 19 é ilustrado o fluxo dos *logs* coletados pelo Logstash.

Figura 19 – Fluxo de dados coletados pelo Logstash



Fonte: Logstash, 2015

Sua utilização permite que todos os logs sejam coletados sem a necessidade de implementar um mecanismo complexo para isso. Mas, assim como o Elasticsearch, o Logstash roda de forma independente. Isto significa que ele também apresenta como desvantagem o consumo extra de recursos.

3.5 Implementação dos serviços

Os serviços do *backend* foram idealizados de forma abstrata. Para desempenhar suas funções como esperados eles serão concretizados na forma de um ou mais microserviços, utilizando pilhas tecnológicas específicas para as necessidades de cada serviço.

3.5.1 Microserviço de produtos

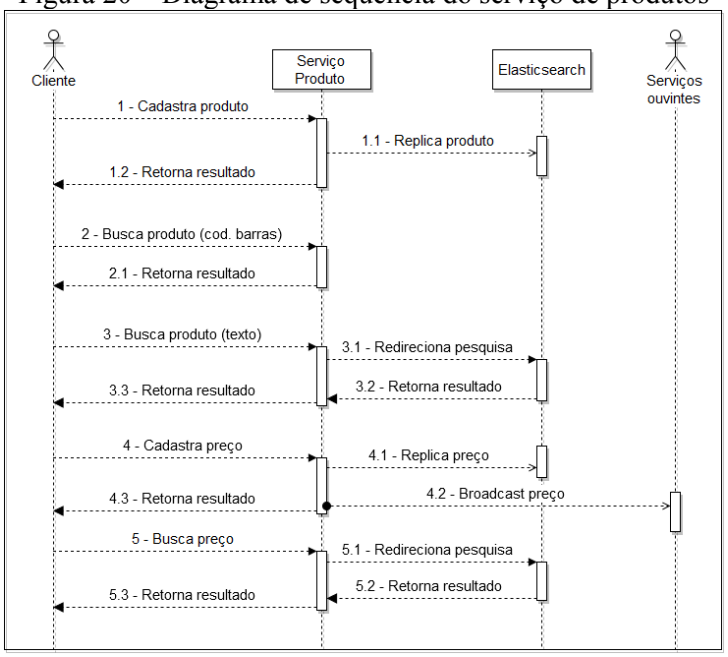
Este microserviço implementa parte das funcionalidades delegadas ao serviço de produtos. Ele é implementado utilizando o Play Framework com a linguagem Scala. Sua camada de acesso ao banco utiliza o Slick. Suas principais atribuições estão relacionadas ao armazenamento e busca de informações. Seus dados serão armazenados em um banco de dados

PostgreSQL próprio, mas também são replicados no Elasticsearch.

O microserviço de produtos trabalha em conjunto com o Elasticsearch para efetuar a busca de produtos por texto e a busca georreferenciada de preços.

O diagrama de sequência apresentado na Figura 20 ilustra o funcionamento deste serviço e dá uma visão geral das principais funções que ele oferece.

Figura 20 – Diagrama de sequência do serviço de produtos

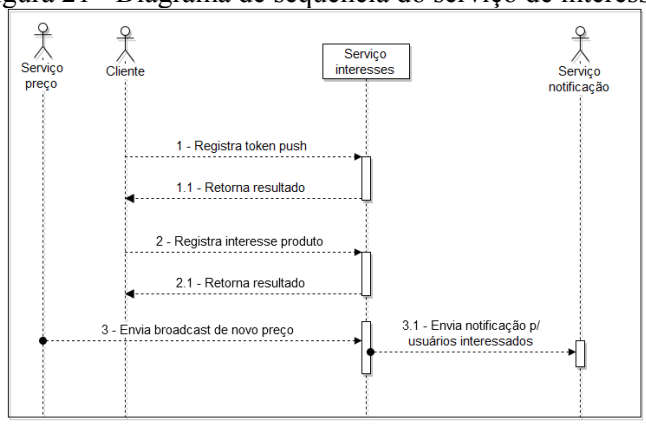


Fonte: Elaborada pelo autor, 2015

3.5.2 Microserviço de interesses

Este microserviço implementa as funcionalidades delegadas ao serviço de interesses. Assim como o microserviço de produtos, o microserviço de interesses foi desenvolvido utilizando o Play Framework com a linguagem Scala e utilizará o PostgreSQL em conjunto com o Slick para o armazenamento e acesso aos dados. O diagrama de sequência apresentado na Figura 21 ilustra o funcionamento deste serviço e dá uma visão geral sobre suas funcionalidades.

Figura 21 - Diagrama de sequência do serviço de interesses

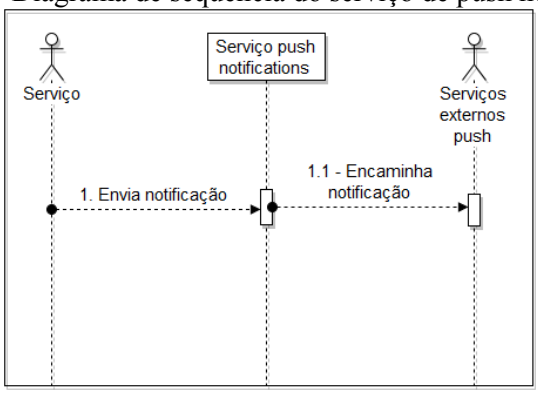


Fonte: Elaborada pelo autor, 2015

3.5.3 Microserviço de push notifications

Este microserviço implementa as funcionalidade delegadas ao serviço de *push notifications*. Ele foi desenvolvido utilizando o Play Framework com a linguagem Java. O diagrama de sequência apresentado na Figura 22 ilustra o funcionamento deste serviço.

Figura 22 - Diagrama de sequência do serviço de push notifications

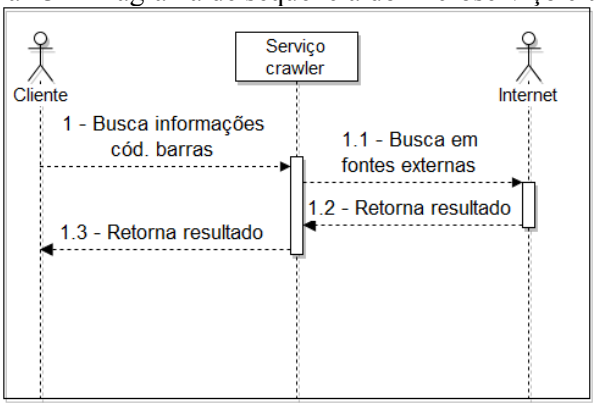


Fonte: Elaborada pelo autor, 2015

3.5.4 Microserviço *crawler*

Este microserviço implementa as funcionalidade delegadas ao serviço *crawler*. Ele foi desenvolvido utilizando o Play Framework com a linguagem Java. O diagrama de sequência apresentado na Figura 23 ilustra o funcionamento deste serviço.

Figura 23 – Diagrama de sequência do microserviço crawler



Fonte: Elaborada pelo autor, 2015

3.5.5 Microserviço de monitoramento

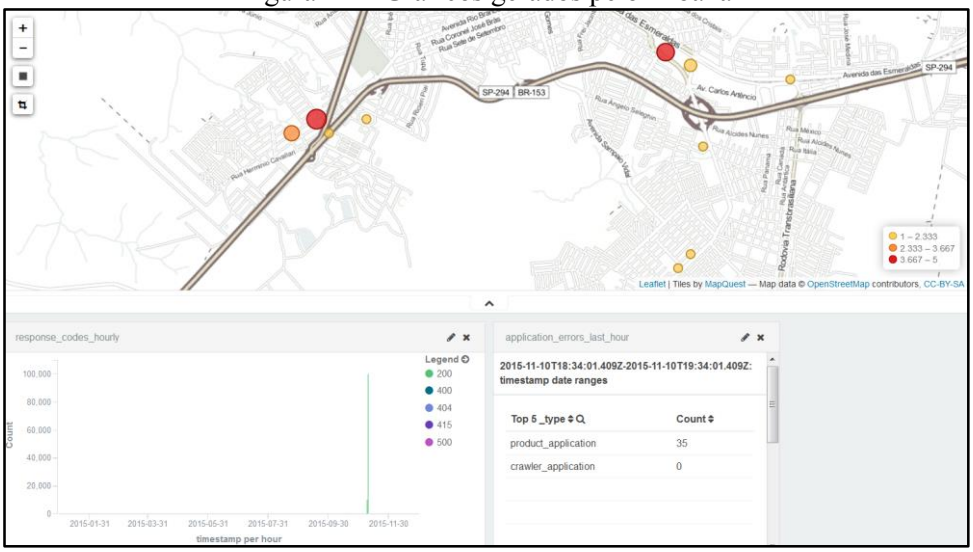
Este serviço implementa as funcionalidades delegadas ao serviço de monitoramento. Ele utiliza um conjunto de três ferramentas para realizar suas funções: Logstash, para coletar e transformar arquivos de *log*, Elasticsearch para armazenar, indexar e disponibilizar na forma de documentos JSON os *logs* coletados pelo Logstash e Kibana para visualizar os dados armazenados no Elasticsearch por meio de uma interface amigável.

Depois de indexados os dados poderão ser acessados de forma simplificada pelo Kibana. O Kibana é uma plataforma de analíticas e visualização que permite que dados armazenados no Elasticsearch sejam resumidos e transformados em gráficos em tempo real. Ele foi desenvolvido para trabalhar em conjunto com o Elasticsearch, permitindo que dados estruturados e não-estruturados ganhem forma. Diversos tipos de gráficos como barras, histogramas, dispersão e até mapas de calor podem ser criados de forma simplificada no Kibana a partir dos dados disponibilizados no Elasticsearch (KIBANA, 2015).

Na Figura 24 são apresentados alguns dos gráficos que podem ser gerados pelo Kibana:

- Um mapa de calor que permite visualizar a concentração das localizações dos preços que foram cadastrados na aplicação.
- Gráfico dos códigos de resposta HTTP gerados pela aplicação em um espaço de tempo.
- Número de erros gerados pela aplicação em um espaço de tempo

Figura 24 – Gráficos gerados pelo Kibana



Fonte: Elaborado pelo autor, 2015

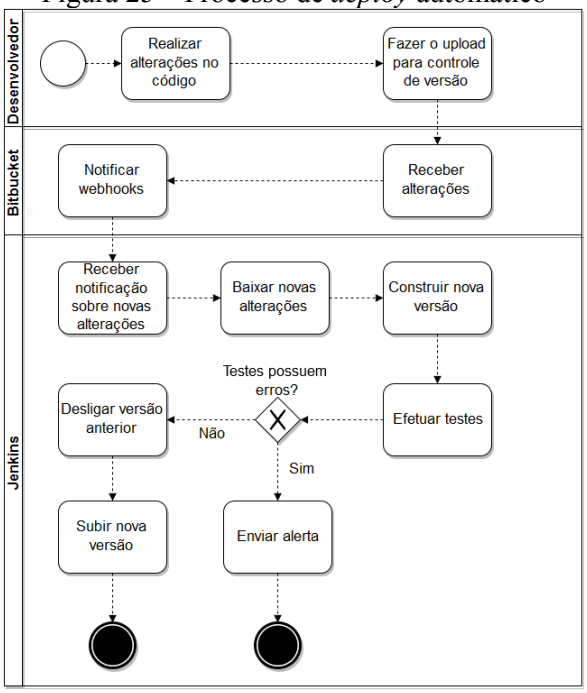
Este serviço é estratégico para todo o projeto proposto, já que ele garante um dos pontos chave no sucesso da implementação de uma arquitetura de microserviços: o monitoramento em tempo real de todo o ecossistema.

3.6 Automação do *deploy*

Para automatizar o processo de testes e *deploy* dos microserviços, foi utilizado o Jenkins. O Jenkins é um servidor open-source de integração e entrega contínua que tem como principal objetivo aumentar a produtividade. Ele pode ser utilizado para fazer builds e testes de forma contínua a fim integrar facilmente as mudanças feitas em projetos por desenvolvedores (JENKINS, 2015).

O Bitbucket, serviço que foi utilizado para hospedar o código fonte dos microserviços implementados, notifica o Jenkins sempre que novas alterações são disponibilizadas. Assim o Jenkins pode, de forma quase instantânea, realizar testes e fazer o *deploy* de novas versões dos microserviços. Na Figura 25 é ilustrado o processo de *deploy* automático que foi criado.

Figura 25 – Processo de *deploy* automático



Fonte: Elaborado pelo autor, 2015

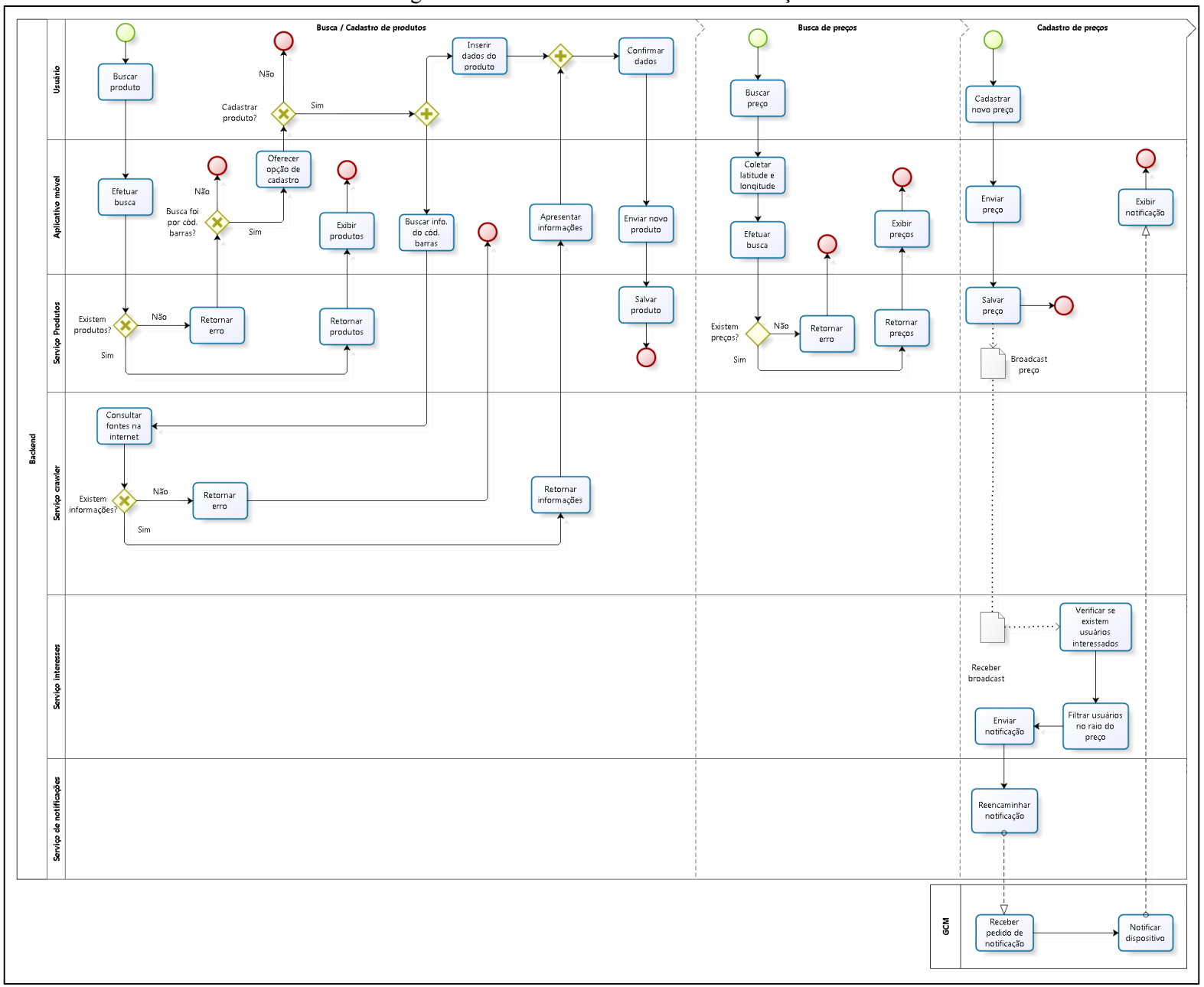
A adoção do processo de automação do *deploy* é importante para garantir que o foco esteja no desenvolvimento dos serviços e que mudanças neles possam ser disponibilizadas de forma rápida para os consumidores finais.

3.7 Dependência entre serviços

O fato da arquitetura de microserviços estar dividida em vários processos individuais possibilita que eles também possam falhar de forma individual. Isso levanta questões sobre como estas falhas individuais se manifestam e como elas influenciam outros processos na arquitetura. É necessário saber como as falhas se propagam pelo sistema e como elas influenciam a experiência do usuário. Entender e modelar as dependências entre os serviços é parte crucial ao entender a inter-dependência entre serviços neste tipo de arquitetura (UHLE, 2014).

O processo ilustrado na Figura 26 dá uma visão geral sobre o fluxo dos processos dentro da arquitetura, e permite identificar os níveis de dependência entre os serviços e consumidores.

Figura 26 –Processo interno dos microserviços



Fonte: Elaborado pelo autor, 2015

3.8 Desafios e problemas no desenvolvimento da proposta

O desenvolvimento da proposta apresentou diversos desafios. Um deles foi a separação da aplicação em serviços antes da implementação, principalmente pelo fato de não existir nada implementado durante esta etapa. Os principais casos de sucesso na implantação de arquiteturas baseadas em microserviços, inclusive os citados no item 2.6, partem de aplicações existentes. Isto facilita a separação em serviços, já que os desenvolvedores conhecem todo domínio de sua arquitetura, funcionalidades que podem ser isoladas e possíveis gargalos.

Outro desafio foi determinar a forma que os microserviços interagiriam entre si. É uma das tarefas mais difíceis, já que a comunicação é um ponto chave dos microserviços, e a utilização de técnicas inadequadas pode comprometer a evolução de toda a arquitetura.

Definir como o cliente se comunica com todos os serviços também foi desafiador. A utilização de um único ponto de entrada para toda a arquitetura poderia se tornar um problema, já que este pode ser visto como um possível gargalo em cenários de alta demanda. A escolha de uma ferramenta adequada para a solução do problema foi crucial para evitar problemas futuros.

3.9 Considerações finais do capítulo

Neste capítulo foi apresentado o desenvolvimento da proposta - uma arquitetura baseada em microserviços. Como resultado, foi gerado um aplicativo móvel para a plataforma Android e um *backend* dividido em cinco microserviços, três deles a serem consumidos ou alimentados pelo aplicativo móvel. Esta divisão interna foi abstraída através da utilização de um Gateway de APIs.

Foi utilizado um processo de *deploy* automático para lançar alterações de forma mais rápida e permitir que o foco do desenvolvedor esteja na implementação e não na infraestrutura. Também foi criado um microserviço que permite monitorar em tempo real o estado de toda a arquitetura.

Espera-se que a arquitetura implementada tire proveito dos benefícios que a arquitetura de microserviços oferece.

4 RESULTADOS

Este projeto teve como resultado o desenvolvimento de uma aplicação composta por um *backend*, dividido em cinco serviços que foram implementados com a utilização de diferentes tipos de tecnologias, e um aplicativo móvel que alimenta e consome dados para a aplicação principal.

Neste capítulo serão apresentados o aplicativo móvel, os testes realizados para avaliar o comportamento da arquitetura e a avaliação sobre a arquitetura baseada nos resultados obtidos nos testes.

4.1 Aplicativo móvel (cliente)

O aplicativo móvel foi desenvolvido com o intuito de testar as funcionalidades oferecidas pela arquitetura proposta. Ele fornece um meio de alimentar e consumir dados para o *backend*.

O aplicativo foi desenvolvido de forma nativa para a plataforma Android, utilizando a linguagem de programação Java.

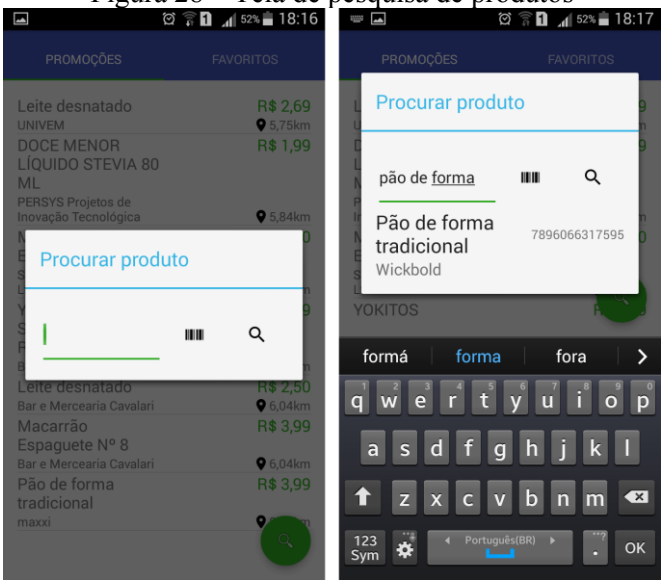
Ao abrir o aplicativo, uma tela com duas abas é apresentada ao usuário: promoções e favoritos, como ilustrado na Figura 27. Na aba de promoções o aplicativo mostra preços que foram marcados como promocionais em estabelecimentos próximos ao usuário. Na aba de favoritos, o usuário acompanha o preço mínimo e máximo de produtos que ele sinalizou como favorito.

Figura 27 – Tela inicial do aplicativo

Fonte: Elaborado pelo autor, 2015

A partir da tela principal, o usuário pode iniciar a busca por um produto ao clicar no botão de pesquisa. Na tela de pesquisa, o usuário tem a opção de buscar por palavras chave relacionadas ao produto, ou escanear um código de barras, como ilustrado na Figura 28. O usuário pode então selecionar um produto para visualizar seus detalhes e preços em estabelecimentos próximos.

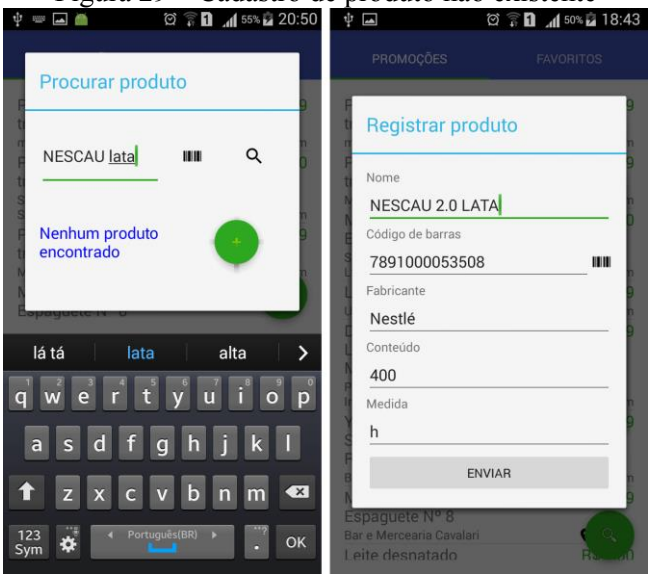
Figura 28 – Tela de pesquisa de produtos



Fonte: Elaborado pelo autor, 2015

Se o usuário efetuar uma busca por um produtos e esta não trazer resultados, ele tem a opção de cadastrar um novo produto. Se a busca foi feita utilizando o código de barras, o aplicativo tentará buscar na Internet informações sobre ele através do serviço *crawler* e sugeri-las ao usuário. Caso contrário, os dados são inseridos normalmente pelo usuário, como ilustrado na Figura 29.

Figura 29 – Cadastro de produto não existente



Fonte: Elaborado pelo autor, 2015

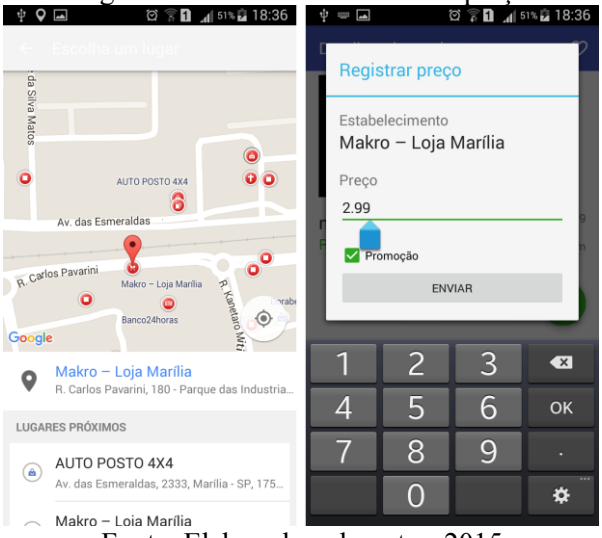
Após efetuar uma pesquisa e selecionar um produto, o usuário é levado a tela de detalhes do produto. Nesta tela são apresentadas informações sobre o produtos e também uma lista com preços deste produto em estabelecimentos próximos ao usuário, como ilustrado na Figura 30. Ainda nesta tela, o usuário pode marcar o produto como favorito ou cadastrar um novo preço.

Figura 30 – Tela de detalhes do produto

Fonte: Elaborado pelo autor, 2015

Ao clicar no botão para cadastrar um novo preço, o usuário é levado a uma tela para selecionar a qual estabelecimento o preço se refere. Após selecionar o estabelecimento, o usuário digita o preço do produto e sinaliza se este é promocional. O processo para cadastrar preços está ilustrado na Figura 31.

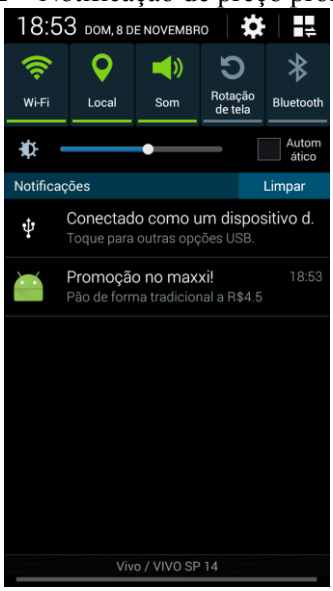
Figura 31 – Telas de cadastro de preço



Fonte: Elaborado pelo autor, 2015

Quando um dos produtos marcados como favorito pelo usuário tem um preço promocional cadastrado em um estabelecimento próximo a ele, uma notificação é enviada, como ilustrado na Figura 32.

Figura 32 – Notificação de preço promocional



Fonte: Elaborado pelo autor, 2015

4.2 Desempenho dos microserviços

Uma das propostas do trabalho é melhorar a escalabilidade da aplicação através do uso dos microserviços. Para avaliar como a arquitetura se comporta em um cenário de estresse, foi criado um cenário para testes. Este cenário tem como objetivo avaliar duas características da arquitetura de microserviço: os efeitos da comunicação entre os serviços e a capacidade da arquitetura de se manter estável mesmo com alta demanda.

4.2.1 Ambiente de teste

Para efetuar os testes, foi criado um ambiente cloud no Azure. O Microsoft Azure fornece soluções PaaS e IaaS para cloud computing, através de uma rede global de datacenters próprios e de parceiros (AZURE, 2015).

Servidor de aplicação

Os microserviços foram alocados em um único servidor para testes, para avaliar se eles poderiam ser afetados devido à sobrecarga na rede. A máquina virtual utilizada para hospedar os microserviços tem as seguintes configurações:

- Processador AMD Opteron 4171 HE de 4 núcleos
- 7 GB de memória
- HD de 30GB
- OpenLogic 7.1 como sistema operacional

Servidor de testes

Os testes foram realizados a partir de uma máquina virtual com as seguintes configurações:

- Processador AMD Opteron 4171 HE com 1 núcleo
- 1,75GB de memória
- HD de 30GB
- OpenLogic 7.1 como sistema operacional

As máquinas virtuais foram alocadas na mesma rede local, para evitar influência de fatores externos.

4.2.2 Ferramentas de testes

Para gerar carga no servidor de aplicação a partir do servidor de testes, foi utilizado o JMeter. O JMeter é uma ferramenta projetada para testar o comportamento funcional e medir performance de aplicações. Ele pode ser utilizado para testar a performance de recursos estáticos e dinâmicos (*webservice*s SOAP/REST), linguagens para a web (Java, PHP, ASP.Net, etc) e também pode ser usado para simular uma grande carga em um servidor ou grupo de servidores (JMETER, 2015).

Pelo fato do JMeter somente coletar dados da perspectiva do cliente, foi utilizado em conjunto com ele a ferramenta de monitoramento New Relic, que permite coletar dados da perspectiva do servidor. O New Relic oferece um conjunto de ferramentas de software utilizadas por desenvolvedores, *ops* e empresas de software para entender como aplicações estão se comportando em ambientes de desenvolvimento e produção (NEWRELIC, 2015).

4.2.3 Testes de carga

Foram realizados quatro casos de testes no JMeter, simulando diferentes tipos de cargas nos serviços:

- 3 clientes realizando um número infinito de requisições para 3 microserviços simultaneamente por 5 minutos seguidos.
- 30 clientes realizando um número infinito de requisições para 3 microserviços simultaneamente por 5 minutos seguidos.
- 300 clientes realizando um número infinito de requisições para 3 microserviços simultaneamente por 5 minutos seguidos.
- 3000 clientes realizando um número infinito de requisições para 3 microserviços simultaneamente por 5 minutos seguidos.

Os microserviços que foram avaliados nos testes foram o de produtos, o de interesses e o de notificações. As seguintes funcionalidades foram utilizadas:

- Microserviço de produtos - Listar promoções em uma região: escolhida pelo fato de exigir que o microserviço se comunique com o Elasticsearch.
- Microserviço de *push notifications* - Enviar *push notifications*: escolhida por funcionar através do método POST, possuir conteúdo no corpo da requisição e pela necessidade da comunicação com uma API externa.
- Microserviço de interesses - Escolhida por retornar uma quantidade razoável de dados na resposta.

Todos os testes foram realizados três vezes. Os valores obtidos são o resultado de uma média de cada grupo de testes.

4.2.4 Métricas

Foram definidas algumas métricas com o objetivo de comparar como a arquitetura se comporta diante dos casos de testes propostos.

Latência média: O tempo que uma requisição demora para ir e voltar do servidor. A latência pode ser influenciada por fatores externos, como carga extra na rede.

Throughput: Quantidade máxima de requisições que a aplicação consegue aceitar em um determinado espaço de tempo.

Taxa de erro: Porcentagem de requisições que não foram respondidas pela aplicação. Erros podem pode ser causados por fatores como tempo de limite excedido, erros internos do servidor, entre outros.

4.2.5 Resultados da latência média

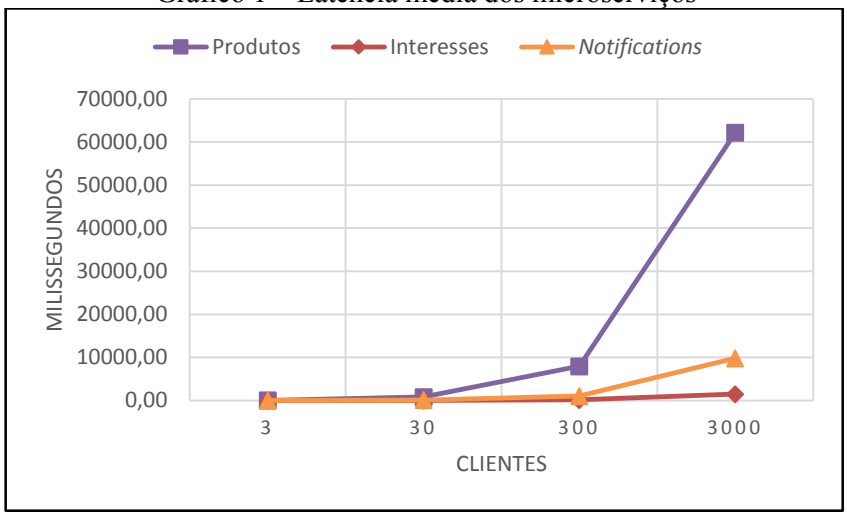
Os testes geraram os seguintes resultados referentes à latência média:

Tabela 2 – Resultado dos testes de latência média, medida em milissegundos (ms)

Nº de clientes	Produtos	Interesses	*Notifications*
3	45,63	5,57	10,08
30	817,89	15,22	104,17
300	7.961,69	154,42	1.060,19
3000	62.168,14	1.476,187	9.796,7

Os dados apresentados na Tabela 2 e no Gráfico 1 foram coletados pelo JMeter. Eles seguem um padrão: de um forma geral, a latência média aumenta na mesma magnitude que o número de clientes simultâneos.

Gráfico 1 – Latência média dos microserviços

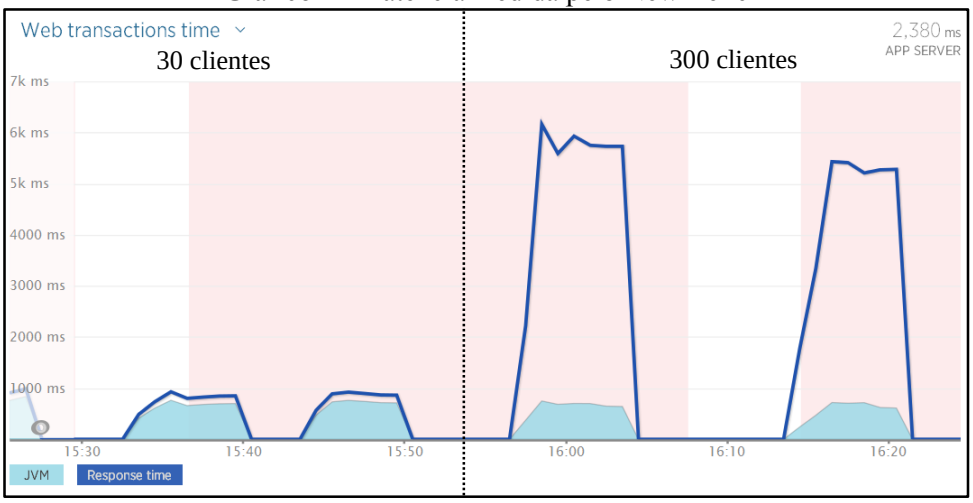


A partir destes dados foi identificado que o microserviço de produtos foi o mais afetado pela alta carga aplicada.

Ao cruzar os dados gerados pelo JMeter com as informações de *web transactions* do New Relic, apresentadas no Gráfico 2, é possível entender o aumento da latência pela perspectiva do servidor.

O intervalo de tempo ilustrado no Gráfico 2 aconteceu entre os testes com 30 clientes e 300 clientes no microserviço de produtos. Neste gráfico é possível identificar que durantes os dois testes, as requisições passaram o mesmo tempo dentro da JVM sendo processados. Apesar disto, os testes com 300 clientes tiveram uma latência muito maior que os testes com 30 clientes.

Gráfico 2 – Latência medida pelo New Relic



Causa:

Estes resultados foram causados pelo fato de o microserviço de produtos ter sido implementado sem seguir as melhores práticas recomendadas pelo Play Framework.

Para entender como o Play Framework trabalha, é interessante compará-lo ao modelo de uma *thread* por requisição. Neste modelo, as requisições ficam enfileiradas numa *pool* de *threads*. Quando houver uma *thread* disponível, a requisição é executada e quando esta for terminada, a *thread* é liberada novamente para a *pool* para ser usada em uma nova requisição.

Mas o Play Framework trabalha de forma diferente, adotando um modelo assíncrono. Uma única requisição passa por diferentes *threads*. Uma fila para uma *pool* de *threads* não contém somente requisições esperando para serem executadas. Ela pode contar requisições que estão no meio do seu tempo de vida, diferentes tarefas que fazem parte de uma mesma requisição ou até mesmo uma tarefa de várias requisições diferentes. Para tirar proveito da arquitetura do Play, aplicações que usam código bloqueante precisam ser projetadas para usar corretamente suas diferentes *pools* de *threads* (PLAY FRAMEWORK, 2015).

Problema e solução:

O problema do microserviço de produtos foi identificado como sendo um trecho bloqueante no código que fazia consultas ao Elasticsearch. Este trecho utilizava uma única *thread* para processar a requisição e para consultar o Elasticsearch e esperar sua resposta. Isso fazia com que as *threads* destinadas às requisições ficassem muito tempo ocupadas, atrasando outras requisições que estavam na fila.

A solução adotada foi separar o trecho de código bloqueante, criando uma *pool* de *threads* dedicadas a efetuar consultas no Elasticsearch e esperar suas respostas. Isto permitiu que parte do trabalho das *threads* na *pool* de requisições fosse aliviado e também implicou na

criação de uma *pool* dedicada, o que permite que ela seja tratada de forma diferenciada das outras *pools*, alocando mais núcleos do processador, por exemplo.

Os testes com 3000 clientes simultâneos foram repetidos para analisar o impacto na latência. Na Tabela 3 são apresentados os resultados dos testes no microserviço de produtos após as correções.

Tabela 3 – Latência antes e depois da correção, medida em milissegundos (ms)

Número de clientes	Produtos (bloqueante)	Produtos (não-bloqueante)	Diferença
3000	62.168	3.164	59.004 (-95%)

No Gráfico 3 e na Tabela 3 é possível visualizar a melhora no cenário de testes com 3000 clientes simultâneos. A latência média reduziu de uma média de 62.168ms nos primeiros testes para 3.000ms nos testes após as correções no código, uma melhora de 95%.

Gráfico 3 – Latência após refatoração do código bloqueante

A arquitetura de microserviços contribuiu para a rápida solução do problema identificado. O tamanho reduzido do código presente nos serviços permite que problemas sejam encontrados com mais facilidade e a independência dos serviços permitiu que uma correção fosse aplicada no microserviço de produtos, sem afetar os outros microserviços.

4.2.6 Resultados de *throughput*

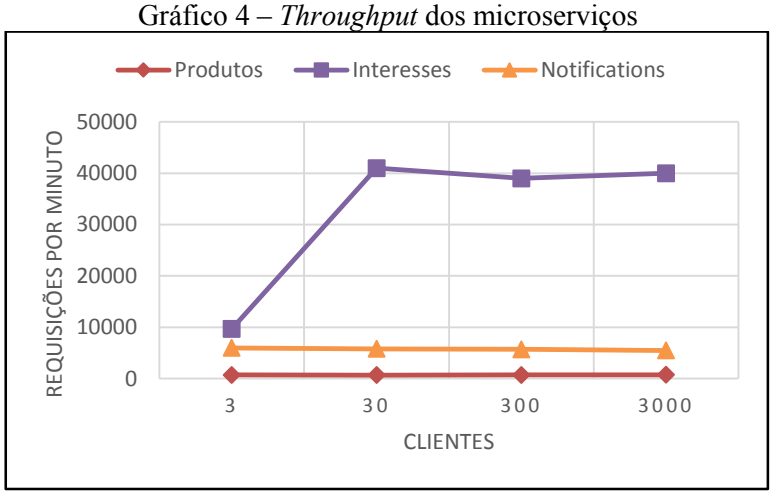
Os dados apresentados na Tabela 4 e no Gráfico 4 foram coletados pelo New Relic. É possível notar que, de forma geral, os microserviços mantêm um *throughput* estável, mesmo com cargas variadas de trabalho.

Tabela 4 – *Throughput* dos microserviços, medido em requisições por minuto (rpm)

Nº de clientes	Produtos	Interesses	*Notifications*
3	707	9.700	6.000
30	690	41.000	5.800
300	728	39.000	5.700
3000	746	40.000	5.500

Os resultados referentes ao *throughput* do serviço de produtos foram coletados antes das correções apresentadas no item 4.2.4. As alterações não afetariam os resultados, já que as requisições que chegam no serviço são tratados por uma *pool* de *threads* diferenciada.

Gráfico 4 – *Throughput* dos microserviços



Esta é outra característica do Play Framework, utilizado na construção dos três microserviços. Ainda que não existam *threads* disponíveis para processar as requisições, os microserviços aceitam todas as requisições que chegam até eles. Elas são alocadas em uma fila até que existam *threads* disponíveis no *pool* de requisições para processá-las. Isso permite que eles mantenham um *throughput* estável, mesmo que as *threads* de requisições estejam ocupadas.

E, como mencionado no item 4.2.5, se forem adotadas boas práticas no gerenciamento das *pools*, ainda é possível projetar aplicações de forma que o *pool* de requisições tenha seu trabalho reduzido.

4.2.7 Resultados da taxa de erros

Na Tabela 5 é possível verificar que a taxa de erro dos microserviços é baixíssima. Isso se deve em parte pela natureza dos testes que foram aplicados. Pelo fato das requisições geradas pelo JMeter serem estáticas, parte delas podem entrar em cache na JVM ou no banco, reduzindo a chance de erros.

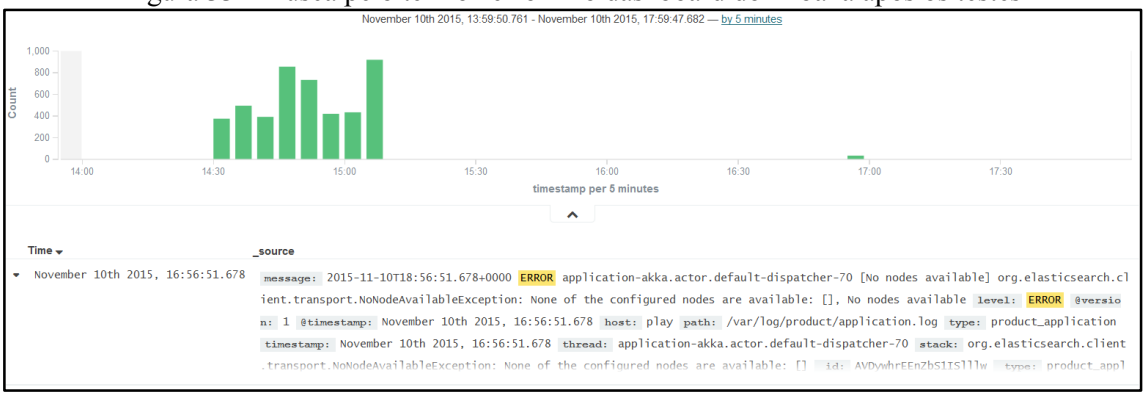
Tabela 5 – Taxa de erro dos microserviços. Porcentagem relativa ao total de requisições.

Nº de clientes	Produtos	Interesses	*Notifications*
3	1%	0%	0%
30	0%	0%	0%
300	0%	0%	0%
3000	0,01%	0%	0%

Apesar da taxa de erros ser baixa, é importante entender suas causas. Através do *dashboard* do Kibana, que faz parte do microserviço de monitoramento, foi possível identificar o que causou os erros gerados pelo microserviço de produtos.

Na Figura 33 é apresentada uma busca pelo termo “error” feita no Kibana. Ele permite visualizar em um gráfico o número de ocorrências nos arquivos de *log* em um espaço de tempo. É possível também visualizar o log original gerado pela aplicação.

Figura 33 – Busca pelo termo “error” no dashboard do Kibana após os testes



Fonte: Elaborado pelo autor, 2015

Graças ao conjunto de ferramentas disponibilizadas pelo microserviço de monitoramento, foi possível identificar rapidamente a causa dos erros: falhas de comunicação entre o microserviço de produtos e o Elasticsearch.

O monitoramento é importante para qualquer tipo de aplicação. Para a arquitetura de microserviços ele é essencial, já que é preciso acompanhar o estado de diversos serviços diferentes e detectar falhas.

4.3 Contribuições

O *backend* foi implementado utilizando o conceito de microserviços, ou seja, todos os serviços que o compõe foram projetados para realizar um conjunto limitado de tarefas, reduzindo sua complexidade. Estes serviços também rodam de forma independente, mantendo

um nível de independência entre si.

Os testes revelaram que a arquitetura proposta se mantém estável em situações de alta demanda e apresenta baixa taxa de falhas. Ela apresenta vantagens e desvantagens.

4.3.1 Vantagens

- Base de código pequena: O maior microserviço implementado foi o de Produtos, que conta com aproximadamente 1900 linhas de código. O tamanho reduzido da base de código faz com que seja fácil conhecer os detalhes da implementação de cada microserviço, acelerando manutenções, por exemplo.
- Fácil de escalar: As funcionalidades estão divididas entre os microserviços. Isso significa que é fácil identificar gargalos de desempenho. Se necessária, a replicação é simples e não exige tantos recursos, já que os microserviços podem ser replicados de forma individual.
- Fácil *deploy*: Adicionar novas funcionalidades e disponibilizá-las é uma tarefa corriqueira com a utilização dos microserviços. O processo de *deploy* automatizado que foi adotado permite que novas funcionalidades estejam disponíveis para consumo dois minutos após o desenvolvedor tê-las disponibilizado no servidor de controle de versão.
- Resiliência: Falhas nos microserviços não se propagam pela aplicação toda. Se um microserviço parar, somente suas funcionalidades ficarão indisponíveis.

4.3.2 Desvantagens

- Complexidade da arquitetura: Conforme novos serviços são adicionados, a complexidade da arquitetura aumenta. É preciso criar estratégias para fazer o *deploy* de todos os serviços, criar mecanismos para detectar e recuperar falhas, criar mecanismos para balanceamento de carga, entre outros.
- Complexidade de um sistema distribuído: Separar as funcionalidades em diversos microserviços significa a introdução da necessidade de comunicação entre eles. Isso traz diversos problemas como a latência da rede, tolerância a falhas, serialização de mensagens, redes não confiáveis, assincronismo, etc.
- Problemas da comunicação assíncrona: na arquitetura proposta, não foi

implementado um mecanismo que garante a entrega de mensagens. Isso significa que partes do sistema ou negócio podem ser afetadas se algum serviço falhar e suas mensagens não forem entregues.

- Desenvolvedores experientes e atualizados: a arquitetura de microserviços permite que tecnologias adequadas sejam aplicadas para resolver problemas específicos, promovendo uma ampla pilha tecnológica. Para isto, é necessário que os desenvolvedores sejam experientes, atualizados e que tenham domínio sobre as soluções aplicadas.

4.4 Lições aprendidas

O desenvolvimento de uma arquitetura na forma de microserviços trouxe uma grande complexidade. Três pontos contribuíram para o sucesso da sua implementação.

***Deploy*:**

Adotar um mecanismo de *deploy* foi fundamental para manter o ciclo de desenvolvimento rápido. A criação de um *pipeline* de entrega contínua permitiu que a adição e disponibilização de funcionalidades se tornassem tarefas comuns. Os esforços foram focados em implementar funcionalidades, e não em gerenciar os microserviços e sua infraestrutura.

Testes:

A atividade de testes é importante para o desenvolvimento de qualquer tipo de software, e neste projeto ela foi fundamental. Diferentes tipos de testes garantem que aplicações atendem requisitos funcionais e não-funcionais. A partir dos testes realizados na arquitetura implementada foi possível detectar e corrigir falhas na implementação.

Monitoramento:

A utilização de ferramentas de monitoramento permite acompanhar em tempo real todo o ecossistema de serviços. Este tarefa é de extrema importância quando existe um grande número de serviços e é necessário identificar quando eles falham e a causa das falhas.

Todo o trabalho desenvolvido contribuiu para o conhecimento sobre a história das arquiteturas modernas, e um conhecimento mais aprofundado sobre a arquitetura de microserviços. O estudo e a implementação do projeto colaboraram para o estudo e aprendizado

de diversas técnicas e tecnologias que podem ser utilizadas em conjunto com este estilo arquitetural.

4.5 Considerações finais do capítulo

Neste capítulo foi apresentado o aplicativo criado para consumir as funcionalidades dos microserviços, os resultados dos testes realizados para avaliar o comportamento da arquitetura em cenários de alta carga e as vantagens e desvantagens da arquitetura proposta.

O aplicativo foi desenvolvido como um cliente com o objetivo de testar as funcionalidades oferecidas pelo *backend*. Este cliente poderia estar em diversas outras formas, como uma aplicação web ou *desktop*. As funcionalidades oferecidas pelo aplicativo atenderam as expectativas, e correspondem às funcionalidades oferecidas pelo *backend*.

Os testes revelaram que o desempenho da arquitetura proposta é extremamente satisfatório. A vazão dos microserviços se mantém constante e a taxa de erros baixa, mesmo com um número alto de clientes simultâneos.

Eles também revelaram a existência de problemas na latência em cenários de alta carga. As características da arquitetura permitiram que a causa do problema fosse identificada e corrigida de forma rápida, fazendo com que novos testes apresentassem melhores resultados.

CONCLUSÕES

O objetivo deste trabalho foi implementar uma arquitetura com funcionalidades comuns em aplicações modernas e que fosse baseada em microserviços, a fim de explorar as vantagens e desvantagens que este modelo arquitetônico oferece.

A implementação foi composta por um aplicativo móvel para a plataforma Android - responsável por oferecer uma interface com usuários - e um *backend* - responsável por oferecer as funcionalidades consumidas pelo aplicativo móvel.

O maior desafio durante o desenvolvimento do aplicativo móvel foi a integração com o *backend,* que estava dividido em diversos serviços diferentes, que por sua vez respondiam em endereços e portas distintas. O problema foi solucionado com a adoção de um Gateway de APIs, que possibilitou abstrair a divisão interna do *backend*, tornando-a transparente para o aplicativo móvel.

O desenvolvimento dos serviços que compõem o *backend* possibilitou o conhecimento de diversas tecnologias e de formas para adaptá-las a soluções diferentes. Alguns dos serviços foram desenvolvidos do zero, outros foram compostos por ferramentas já existentes e alguns foram um misto de desenvolvimento em conjunto com ferramentas prontas. Isto evidencia a característica da heterogeneidade tecnológica das arquiteturas baseadas em microserviços, que permite aplicar as tecnologias mais adequadas na solução de determinados problemas.

A implementação do *backend* e os testes realizados sobre ele também contribuíram para o conhecimento sobre as vantagens e desvantagens da arquitetura de microserviços.

Conclui-se que a utilização de arquiteturas baseadas em microserviços é uma forma viável para a construção de aplicações corporativas. Ela oferece diversas vantagens se comparada às tradicionais arquiteturas monolíticas, principalmente para grandes aplicações corporativas.

TRABALHOS FUTUROS

A partir do projeto desenvolvido, pode-se destacar como possíveis trabalhos futuros:

- Fazer o *deploy* dos microserviços em máquinas virtuais individuais, realizar testes de carga e avaliar o custo/benefício da migração em relação à utilização de uma instância única
- Adicionar um mecanismo de Machine Learning no microserviço de produtos para determinar a confiabilidade de informações inseridas por usuários. Avaliar o impacto da mudança na arquitetura.
- Testar outros paradigmas para a comunicação entre os serviços como a Descoberta de Serviços.

REFERÊNCIAS BIBLIOGRÁFICAS

AMARAL, Marcelo [*et al.*] "Performance Evaluation of Microservices Architectures using Containers." arXiv preprint arXiv:1511.02043 (2015).

AZURE. Azure Portal, 2015. Disponível em: <https://portal.azure.com/> Acessado em: 16 novembro 2015.

COCKROFT, Adrian. Talking microservices with the man who made Netflix's cloud famous, 2015. Disponível em: < https://medium.com/s-c-a-l-e/1032689afed3> Acessado em: 16 novembro 2015.

ELASTICSEARCH. Revealing Insights from Data, 2015. Disponível em: <https://www.elastic.co/> Acessado em: 16 novembro 2015.

FARNAGHI, M.; Mansourian, A.; Toomanian, A.. Integration of rendering technologies and visualization techniques to improve 3D Mobile GIS applications, 2009.

FIELDING, Roy. Representational state transfer, 2000.

FOWLER, Martin. Patterns of Enterprise Application Architecture, 2003, Addison-Wesley Longman Publishing Co., Inc.

FOWLER, Martin. Microservices, 2014. Disponível em: <http://martinfowler.com/articles/microservices.html> Acesso em: 16 novembro 2015.

FURHT, Borko [*et al*.]. An Innovative Internet Architecture for Application Service Providers, 2000.

GOSLING, James [*et al.*]. The Java language specification, Java SE 8 Edition, 2015.

JENKINS. Jenkins CI, 2015. Disponível em: <https://jenkins-ci.org/> Acessado em: 16 novembro 2015.

JMETER. Apache Jmeter, 2015. Disponível em: <http://jmeter.apache.org/> Acessado em: 16 novembro 2015.

KIBANA. Explore, visualize discover data, 2015. Disponível em: <https://www.elastic.co/products/kibana> Acessado em: 16 novembro 2015.

LOGSTASH. Collect, parse transform logs, 2015. Disponível em: <https://www.elastic.co/products/logstash> Acessado em: 16 novembro 2015.

MICROSOFT CORPORATION, Enterprise Solution Patterns, 2003. Disponível em: <https://msdn.microsoft.com/en-us/library/ff647095.aspx> Acesso em: 16 novembro 2015.

NAMIOT, Dmitry; SNEPS-SNEPPE, Manfred. On micro-services architecture, 2014.

NEW RELIC. Application Performance Management and Monitoring, 2015. Disponível em: <http://newrelic.com/> Acessado em: 16 novembro 2015.

NEWMAN, Sam. Building Microservices, 2015, O'Reilly Media, Inc.

NIELSEN, Claus Djernæs. Investigate availability and maintainability within a microservice architecture, 2015.

PAPAZOGLOU, Mike . Service-oriented computing: Concepts, characteristics and directions, 2003.

PELTZ, Chris. Web services orchestration and choreography, 2003.

PLAY FRAMEWORK. The High Velocity Web Framework For Java and Scala, 2015. Disponível em: < https://www.playframework.com/> Acessado em: 16 novembro 2015.

POSTGRESQL. The official site for PostgreSQL, the world's most advanced open source database, 2015. Disponível em: < http://www.postgresql.org/> Acessado em: 16 novembro 2015.

SCALA. The Scala programming language, 2015. Disponível em: <http://www.scala-lang.org/> Acessado em: 16 novembro 2015.

SLICK. Functional Relational Mapping for Scala, 2015. Disponível em: <http://slick.typesafe.com/> Acessado em: 16 novembro 2015.

TANG, Longji; WEI-TEK, Tsai; DONG, Jing. Enterprise Mobile Service Architecture: Challenges and Approaches, 2013, Service-Driven Approaches to Architecture and Enterprise Integration.

TYK. Open Source API Gateway and API Management Platform, 2015. Disponível em: <https://tyk.io/> Acessado em: 16 novembro 2015.

UHLE, Johan. On Dependability Modeling in a Deployed Microservice Architecture, 2014.

VINH D. Le [*et al.*]. Microservice-based Architecture for the NRDC, 2015.

WORLD WIDE WEB CONSORTIUM. Web services architecture, 2004. Disponível em: <http://www.w3.org/TR/ws-arch/> Acessado em: 16 novembro 2015.