

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

André Stangarlin de Camargo

Uma Abordagem para Testes de Desempenho de Microservices

Florianópolis

2016

André Stangarlin de Camargo

Uma Abordagem para Testes de Desempenho de Microservices

Dissertação submetida ao Programa
de Pós-Graduação em Ciência da Computação
para a obtenção do Grau de
Mestre em Ciência da Computação.
Orientador: Prof. Frank Augusto Siqueira

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Camargo, Andre Stangarlin de
Uma Abordagem para Testes de Desempenho de
Microservices / Andre Stangarlin de Camargo ; orientador,
Frank Augusto Siqueira - Florianópolis, SC, 2016.
95 p.

Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico. Programa de Pós-Graduação em
Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Sistemas Distribuídos. 3.
Web Services. 4. Microservices. 5. Teste de Desempenho. I.
Siqueira, Frank Augusto. II. Universidade Federal de Santa
Catarina. Programa de Pós-Graduação em Ciência da Computação.
III. Título.

André Stangarlin de Camargo

**TÍTULO DO TRABALHO: Uma Abordagem para Testes de
Desempenho de Microservices**

Esta dissertação foi julgada adequada para obtenção do título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 16 de dezembro de 2016.

Prof^ª. Carina Friedrich Dorneles, Dr^ª.
Coordenadora do Programa

Banca Examinadora:

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina
Orientador

Prof^ª. Daniela Barreiro Claro, Dr^ª.
Universidade Federal da Bahia

Prof. Mario Antonio Ribeiro Dantas, Dr.
Universidade Federal de Santa Catarina

Prof^ª. Patrícia Vilain, Dr^ª.
Universidade Federal de Santa Catarina

RESUMO

Em aplicações de grande porte é essencial reduzir o acoplamento entre módulos. Dessa forma, é possível reduzir o impacto das alterações em componentes distintos, bem como aprimorá-los de forma independente. Assim, surgiu o conceito de microservices, apresentado como uma alternativa ao modelo tradicional, conhecido como aplicações monolíticas. O modelo tradicional é criticado devido à difícil manutenção e evolução, ocasionada pelo elevado grau de acoplamento entre os componentes (FOWLER-LEWIS, 2014).

A arquitetura de microservices prevê a separação de uma aplicação em um conjunto de serviços de menor complexidade, cada qual executando de forma independente e utilizando protocolos simples para comunicação, como HTTP (FOWLER-LEWIS, 2014). O modelo vem sendo amplamente utilizado, principalmente pela facilidade na manutenção e evolução das aplicações. A adoção desse modelo de arquitetura acaba por transformar uma única aplicação monolítica em um conjunto de serviços (NEWMAN, 2015), que tende a crescer com a adição de novas funcionalidades.

No âmbito de microservices, existe a necessidade de prover garantias de Qualidade de Serviço (QoS), em relação a requisitos não funcionais como: disponibilidade, confiança, segurança e desempenho (MANI-NAGARAJAN, 2002). Em se tratando especificamente do campo desempenho, é necessário conhecer a capacidade e o tempo de resposta de um serviço para que se possa avaliar melhorias e correções sob a perspectiva dessas métricas.

A proposta do presente trabalho é definir um modelo arquitetural que possa automatizar os testes de desempenho dos serviços em um conjunto de microservices. O problema foi endereçado à arquitetura de microservices em virtude desta representar o contexto no qual o conhecimento da capacidade dos serviços é de extrema importância, sobretudo devido à dinamicidade que os

serviços possuem, sendo que novas mudanças e funcionalidades tendem a alterar a capacidade do serviço.

Com base na arquitetura proposta foi desenvolvido um framework que implementa os conceitos propostos pela arquitetura. A avaliação do framework demonstrou que o mesmo pode ser utilizado sem qualquer prejuízo ao desempenho do serviço.

Palavras-chave: microservices, teste de desempenho, teste automatizado.

LISTA DE FIGURAS

Figura 1 - Aplicação Monolítica X Microservices (FOWLER-LEWIS, 2014). Adaptado.	21
Figura 2 - Esquema representando o funcionamento de um serviço (KRAFIG et al., 2005). Adaptado.	26
Figura 3 - Principais componentes em uma arquitetura orientada a serviços (IBM, 2000).	27
Figura 4 - Microservices com Diferentes Tecnologias (NEWMAN, 2015).	39
Figura 5 – Docker: contêiner e imagens (DOCKER, 2015).	43
Figura 6 – Docker e Máquina Virtual (DOCKER, 2015).	45
Figura 7 - Exemplo de Requisição e Resposta utilizando o SOAPUI.	52
Figura 8 - SOAPUI: Teste de Carga.	53
Figura 9 – JMeter: requisição a um serviço SOAP.	54
Figura 10 - JMeter: Definição dos Parâmetros de Teste.	55
Figura 11 - JMeter: Relatório de Teste.	55
Figura 12 - Visão Geral da Arquitetura Proposta.	62
Figura 13 - Exemplo de Resposta para uma Requisição Options.	64
Figura 14 - Comportamento do Framework para Requisições HTTP.	68
Figura 15 - Uso do Framework para Definir uma Especificação de Teste.	71
Figura 16 - Classe Utilizada para Geração da Especificação de Teste.	72
Figura 17- Exemplo de Erro de Compilação Durante a Validação das Anotações.	75
Figura 18 - Resultados do Teste Piloto.	78
Figura 19 - Resultado para o Primeiro Teste.	84
Figura 20 - Resposta do Serviço para o Primeiro Teste.	84
Figura 21 - Resultado para a Primeira Rodada.	85

LISTA DE QUADROS

Quadro 1 - Exemplo de documento WSDL.....	29
Quadro 2 - Exemplo de envelope SOAP de uma requisição.	300
Quadro 3 - Exemplo de envelope SOAP de uma resposta.	311

LISTA DE ABREVIATURAS E SIGLAS

AUFS	<i>Another Union File System</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IaaS	<i>Infrastructure as a Service</i>
JDK	<i>Java Development Kit</i>
JSON	<i>JavaScript Object Notation</i>
IOPS	<i>Input/Output Operations Per Second</i>
ITU	<i>International Telecommunication Union</i>
NoSQL	<i>Not only SQL</i>
QoS	<i>Quality of Service</i>
REST	<i>Representational State Transfer</i>
SLA	<i>Service Level Agreement</i>
SOAP	<i>Simple Object Access Protocol</i>
WSDL	<i>Web Service Description Language</i>
W3C	<i>World Wide Web Consortium</i>
XML	<i>Extensible Markup Language</i>
URI	<i>Uniform Resource Identifier</i>

Sumário

1.	INTRODUÇÃO	17
1.1.	PROBLEMA DE PESQUISA	18
1.2.	CONTEXTUALIZAÇÃO	19
1.3.	HIPÓTESE	21
1.4.	OBJETIVOS	22
1.5.	ORGANIZAÇÃO DO TRABALHO	22
2.	SERVIÇOS WEB	25
2.1.	Arquitetura Orientada a Serviços	26
2.2.	Web Services	27
2.2.1.	SOAP	28
2.2.2.	REST (<i>Representational State Transfer</i>)	31
2.3.	Qualidade de Serviço (QoS)	33
2.4.	Considerações Finais do Capítulo	34
3.	MICROSERVICES	37
3.1.	Dividir para Conquistar	38
3.2.	Independência Tecnológica	39
3.3.	Resiliência	40
3.4.	Implantação (<i>Deploy</i>)	41
3.4.1.	Ferramenta <i>Docker</i>	42
3.5.	Limitações	45
3.6.	Considerações Finais do Capítulo	48
4.	TESTE DE SOFTWARE	49
4.1.1.	Testes de Desempenho	49

4.1.2.	Testes de Serviços.....	50
4.2.	Ferramentas para Teste em Web Services	51
4.2.1.	Ferramenta <i>SoapUI</i>	51
4.2.2.	Ferramenta <i>JMeter</i>	54
4.3.	Considerações Finais do Capítulo	56
5.	ESTADO DA ARTE	57
5.1.	Trabalhos Correlatos.....	57
5.1.1.	Especificação Automática de Testes em Web Services ..	57
5.1.2.	Ferramenta para Testar a Resiliência de Microservices ..	59
5.1.3.	Arquitetura para Reutilização de Teste de Aceitação em Microservices	59
5.2.	Considerações Finais do Capítulo	60
6.	ARQUITETURA PARA AVALIAÇÃO DO DESEMPENHO DE MICROSERVICES.....	61
6.1.	Visão Geral da Arquitetura.....	61
6.2.	Especificação dos Testes	63
6.3.	Considerações Finais do Capítulo	65
7.	FPTS – <i>FRAMEWORK FOR PERFORMANCE TEST SPECIFICATION</i>	67
7.1.	Implementação.....	68
7.2.	Forma de Utilização.....	70
7.3.	Principais Funcionalidades.....	73
8.	AVALIAÇÃO.....	77
8.1.	Primeira Etapa – Avaliação do Impacto do Framework	77
8.1.1.	Metodologia.....	78
8.1.2.	Configuração e Execução.....	79
8.1.3.	Resultados	79

8.2.	Segunda Etapa – Avaliação da Geração da Especificação de Teste	83
8.2.1.	Execução e Resultados	83
9.	CONCLUSÕES E TRABALHOS FUTUROS	87
	REFERÊNCIAS	91

1. INTRODUÇÃO

A arquitetura de microservices vem atraindo cada vez mais a atenção dos desenvolvedores de software. São muitas as vantagens proporcionadas pelo uso desta arquitetura, dentre elas: desacoplamento das funcionalidades do sistema, desenvolvimento dos módulos de forma independente, escalabilidade distinta para cada módulo, redução de custos, maior confiabilidade dos serviços, refletem fielmente as funcionalidades do sistema e a sua lógica devido a separação clara dos serviços, podem conter diferentes tecnologias e se necessário uma tecnologia pode ser substituída sem grande impacto (SIMONE, 2014).

Apesar da sua recente popularidade e crescente adoção no desenvolvimento de aplicações, há algumas limitações na arquitetura de microservices para as quais existe um apelo para melhorias e soluções. Os principais aspectos negativos acerca da adoção da arquitetura de microservices são (STENBERG, 2014):

- A dificuldade de instalar e controlar múltiplos serviços em diferentes servidores e que devem cooperar;
- Compartilhar funcionalidades semelhantes entre serviços, o que pode acabar por gerar a necessidade de refatoração em múltiplos serviços.

Naturalmente, desenvolver microservices é trabalhar com sistemas distribuídos, porém há uma grande dificuldade já conhecida para realização de testes em sistemas distribuídos (NAMIOT-SNEPS, 2014): é necessária uma coordenação entre os diferentes processos distribuídos. Outro fator que expressa uma dificuldade em modelos de microservices é o controle da capacidade de fluxo entre serviços. As diferenças em desempenho e fatores como a escalabilidade podem ocasionar um descompasso entre serviços que precisam cooperar.

Para serviços que são dependentes entre si, além de simplificar o processo de implantação, é necessário também realizar um monitoramento do conjunto de serviços para os quais existem essa relação de dependência.

1.1. PROBLEMA DE PESQUISA

Há diversos problemas e desafios na utilização da arquitetura de microservices (HOFF, 2015). Para aplicações de grande porte, a quantidade de serviços pode chegar a milhares. O número grande torna a tarefa de implantação (*deploy*) e integração dos serviços bastante árdua. O problema é que uma aplicação monolítica é decomposta em, pelo menos, dezenas de serviços, o que torna o processo de monitoramento complexo, tornando difícil a detecção e o rastreamento de falhas.

Após a instalação dos serviços é necessário manter uma vigilância sobre o funcionamento dos mesmos. Em se tratando de serviços dependentes (ou seja, que funcionam de forma integrada), é necessário detectar os pontos de integração e monitorá-los a fim de identificar a causa de um problema (HOFF, 2014).

Um tipo de monitoramento é o monitoramento sintético, também conhecido como monitoramento ativo, que é aquele que ocorre com base em uma simulação, de forma contínua e com o objetivo de monitorar: funcionalidade, tempo de resposta e disponibilidade (GABRIELSON, 2013). Esse monitoramento provê recursos para detecção de anomalias de forma instantânea bem como fornece uma visão dos recursos do sistema.

Existem diversas soluções que oferecem formas para realizar o monitoramento ativo. O problema é que, na sua totalidade, essas soluções focam em uma abordagem centrada na perspectiva do usuário, ou seja, em interações de usuários. Exemplos de aplicações com essa finalidade são o *New Relic Synthetic*¹ e o *Dynatrace Synthetic Monitoring*². Porém, os *microservices* são serviços, ou seja, são consumidos por aplicações cliente, geralmente através de Web Services sobre o protocolo HTTP como REST (HOFF, 2014). Nesse caso, não é possível realizar uma abordagem centrada no usuário.

A aplicação de parâmetros de QoS a Web Services remete aos seguintes requisitos não funcionais (YANG et al., 2006): desempenho, segurança, disponibilidade e confiança. Diversas abordagens permitem a construção de Web Services levando em consideração parâmetros de QoS. A linguagem WADL (LUDWIG

¹ <http://newrelic.com/synthetics>

² <http://www.dynatrace.com/en/products/synthetic-monitoring.html>

et al., 2003) foi desenvolvida com o intuito de agregar os parâmetros de QoS a operações em Web Services. Outras abordagens como a proposta por RAJENDRAN-BALASUBRAMANIE (2009) permitem auxiliar a busca por Web Services com base em parâmetros de QoS.

As abordagens que envolvem QoS em Web Services partem do princípio que os parâmetros já foram definidos. Nesse sentido, visam prover soluções para problemas posteriores à fase de definição dos parâmetros, como descoberta e composição dos serviços.

A tarefa de estabelecer os parâmetros de um serviço é árdua e normalmente se dá através de processos manuais e *benchmarks* para avaliar a capacidade de um serviço. No contexto de uma arquitetura de microservices, os serviços mudam constantemente. A cada mudança, uma nova avaliação de desempenho do serviço deve ser feita, principalmente para garantir que a adição de um serviço ou uma nova funcionalidade não impactou no desempenho. Ferramentas para avaliação de desempenho, como SoapUI e JMeter, requerem ações manuais e dependentes do usuário e não são integradas diretamente na arquitetura de microservices.

A detecção de gargalos no desempenho é especialmente importante em uma arquitetura de microservices devido à dependência entre diferentes serviços que são compostos por diversas tecnologias. Em um cenário de integração contínua, é essencial que os serviços possam ser monitorados de maneira rápida e eficiente para não prejudicar o ciclo de integração de novas funcionalidades.

1.2. CONTEXTUALIZAÇÃO

A definição da arquitetura de um sistema é uma tarefa essencial e impactante no projeto em diversos níveis (FOWLER, 2003). Uma decisão inadequada pode levar um projeto ao fracasso e a uma grande perda de receita. Além disso, pode fazer com que a evolução de uma aplicação tenha um custo extremamente elevado e torne-se inclusive impraticável.

Há uma busca contínua para melhorar a maneira como os sistemas são desenvolvidos (NEWMAN, 2015). Muitas propostas de arquitetura foram desenvolvidas ao longo das últimas décadas. Em sua grande maioria, surgem como alternativas a dificuldades enfrentadas durante o desenvolvimento.

Aplicações Web são aplicações projetadas para interagir através da web (CONALLEN, 2002). São hospedadas em um servidor e interagem com clientes através de um navegador Web (*browser*) ou aplicativos, empregando o protocolo HTTP. A aplicação recebe requisições de clientes, processa essas requisições e retorna um resultado (NATIONS, 2014). Essa forma de interação é chamada de cliente-servidor. A camada do servidor, onde está armazenada a aplicação, contém um servidor web, que é uma aplicação responsável por executar e gerenciar o ciclo de vida de uma aplicação web. O servidor necessita de uma plataforma de execução com uma grande capacidade de processamento, pois o mesmo deve ser capaz de atender a diversos clientes de forma simultânea (BEAL, 2015).

Uma aplicação web é, normalmente, reduzida a um pacote de distribuição previamente à sua execução no servidor de aplicação. A geração do pacote de distribuição da aplicação visa organizar todos os seus arquivos, módulos e outras bibliotecas, bem como definir particularidades sobre sua execução. Ao processo de instalação do pacote através do servidor de aplicação até a sua execução, dá-se o nome de implantação (*deploy*).

A arquitetura de *microservices* emergiu como tópico juntamente com conceitos como entrega contínua e virtualização sob demanda. Seu surgimento está intimamente ligado à procura por alternativas ao desenvolvimento de aplicações monolíticas, que são construídas como uma unidade única. Para essas aplicações, a alteração em uma parte do sistema tem como consequência a geração de uma nova versão (pacote) e atualização no servidor (*deploy*) (FOWLER-LEWIS, 2014). Outro grande problema nesse tipo de estrutura é a escalabilidade horizontal. Devido à crescente complexidade dessas aplicações, a escalabilidade tende a ter um custo elevado, uma vez que toda a aplicação tem que ser replicada para outros servidores (FOWLER-LEWIS, 2014). Por fim, os problemas apontados tendem a ter um crescimento contínuo à medida que a aplicação cresce e novas funcionalidades são adicionadas.

A Figura 1 apresenta de forma comparativa as diferenças entre aplicações monolíticas e aplicações concebidas empregando o conceito de microservices.

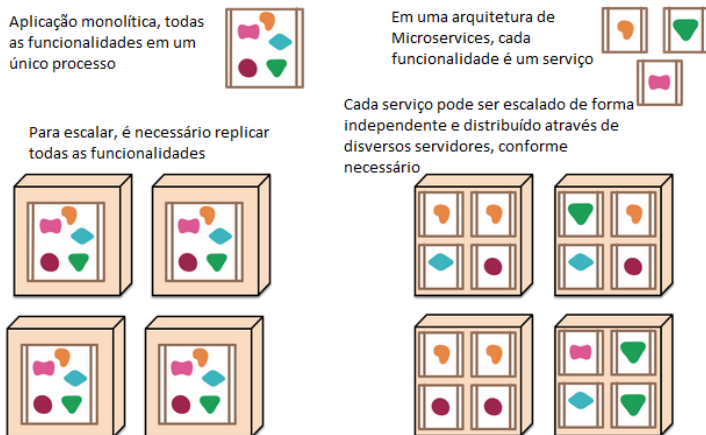


Figura 1 - Aplicação Monolítica X Microservices (FOWLER-LEWIS, 2014). Adaptado.

Como é possível observar, para escalar uma aplicação monolítica é necessário replicar toda a aplicação. Tal procedimento é custoso, uma vez que o consumo de recursos também será replicado. Entretanto, no caso das aplicações em uma arquitetura de microservices, é possível escalar apenas as funcionalidades que realmente necessitam serem escaladas. Essa customização permite definir diferentes níveis de escalabilidade e reduzir os custos com infraestrutura.

1.3. HIPÓTESE

A hipótese assumida é a de que os microservices seguem o estilo arquitetural REST, e portanto, empregam diretamente o protocolo HTTP para comunicação. Os métodos passíveis de avaliação são: POST e GET.

Devido à grande popularidade da ferramenta *Docker* como um contêiner para execução de aplicações, o presente trabalho faz uso

dessa ferramenta para prover a infraestrutura necessária. Essa ferramenta permite também a fácil integração da aplicação a diferentes arquiteturas. Espera-se, no entanto, que a solução proposta possa atender a microservices que não necessitam dessa ferramenta.

1.4. OBJETIVOS

Esta seção apresenta os objetivos do presente trabalho. Os objetivos gerais são as principais contribuições do trabalho e os objetivos específicos são os requisitos que permitem alcançar os objetivos gerais.

1.4.1. OBJETIVOS GERAIS

O presente trabalho tem como objetivo geral o desenvolvimento de uma arquitetura capaz de fornecer mecanismos para avaliar a capacidade de um conjunto de microservices.

1.4.2. OBJETIVOS ESPECÍFICOS

Para realização deste trabalho são almejados os seguintes objetivos específicos:

- Desenvolver um mecanismo para que os serviços possam definir um modelo de requisição para cada uma das suas operações.
- Definir um mecanismo para que as especificações dos testes possam ser acessadas em um único ponto pela aplicação que executa os testes de desempenho.
- Desenvolver um mecanismo para geração da especificação de teste.
- Avaliar a solução através de um experimento real, utilizando modelos de serviços com diferentes complexidades e capacidades.
- Avaliar o impacto da solução proposta no desempenho do serviço.

1.5. ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma: após esta introdução, será apresentada uma revisão bibliográfica acerca dos conceitos de serviços Web, SOA, Web Services, qualidade de

serviço, microservices e teste de software; Em seguida é apresentado o estado arte, onde são apresentados os trabalhos relacionados. A seguir, é apresentada a arquitetura proposta seguida pela apresentação de um framework que implementa alguns dos conceitos propostos pela arquitetura. Em seguida é apresentada uma avaliação do framework. Por fim são apresentadas as conclusões e propostas para trabalhos futuros.

2. SERVIÇOS WEB

Os serviços web (Web Services) estão diretamente relacionados ao conceito de aplicação web. Em um primeiro momento, as aplicações web foram definidas como uma coleção de *servlets*, páginas HTML e outros recursos necessários para a execução da aplicação em um servidor web (RAJIV, 2009). Uma aplicação web pode ser empacotada (*bundle*) e executada em diferentes servidores.

O conceito de aplicações web surgiu juntamente com a especificação dos *servlets*, definido pela Sun³ (CHAFEE, 2012). Os *Servlets*, são classes definidas na linguagem JAVA e que são utilizadas para estender a capacidade de aplicações que operam sobre o paradigma requisição-resposta. Normalmente, estão associados diretamente aos métodos HTTP POST e GET. A especificação dos *servlets* (RAJIV, 2009) traz que os mesmos são componentes gerenciados por um contêiner capaz de gerar conteúdo dinamicamente. Um contêiner é um componente de um servidor web capaz de gerenciar as requisições e respostas durante a interação de um cliente com o servidor. O mesmo é responsável por encapsular processos como: serviços de rede, decodificação das requisições e codificação das respostas.

Um tipo de aplicações web são aquelas orientadas a serviços. Um serviço pode ser entendido como uma operação produzida por um sistema (denominado servidor) e que pode ser acessada através de uma requisição originada por outro sistema (denominado cliente) (KRAFZIG et al., 2005). A Figura 2 mostra como funciona a interação entre um serviço (produtor) e um cliente (consumidor).

O conceito de serviços permitiu o surgimento de um modelo de arquitetura para a concepção de aplicações orientadas a serviços.

³ Sun Microsystems foi a empresa responsável pela criação da linguagem JAVA e diversas tecnologias relacionadas a essa linguagem, dentre elas os *Servlets*.

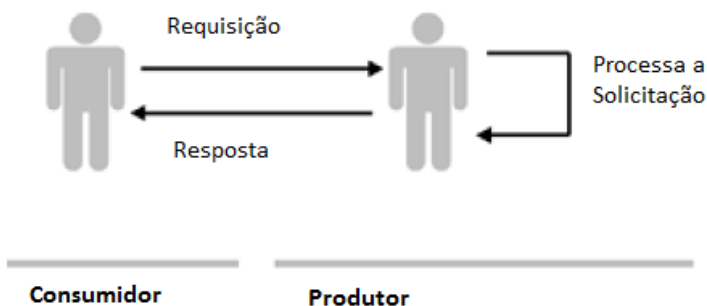


Figura 2 - Esquema representando o funcionamento de um serviço (KRAFIG et al., 2005). Adaptado.

2.1. Arquitetura Orientada a Serviços

Arquitetura orientada a serviços ou SOA (em inglês) é um modelo para construção de aplicações como um conjunto de componentes com baixo grau de acoplamento e que permite a outras aplicações a execução de funções de forma remota (HURWITZ et al., 2007). O conjunto de componentes que forma um serviço pode ser visto como uma “caixa preta”, ou seja, do ponto de vista do cliente, o serviço é acessado através de uma interface que esconde os detalhes acerca do funcionamento do serviço.

O conceito de baixo grau de acoplamento sob o ponto de vista dos serviços está apoiado no fato de que cada serviço deve representar uma regra de negócio bem definida, seguindo um único propósito (HURWITZ et al., 2007). Por exemplo, em uma aplicação de comércio eletrônico, um serviço que calcula as taxas de entrega não deve gerenciar os clientes. O objetivo do baixo acoplamento é facilitar a manutenção e a evolução dos serviços. Cada serviço existe de forma individual e integra-se a outros serviços para prover uma funcionalidade.

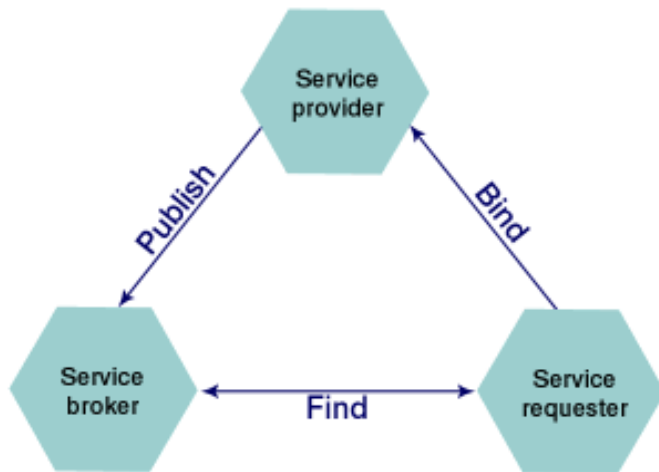


Figura 3 - Principais componentes em uma arquitetura orientada a serviços (IBM, 2000).

O modelo é composto por três tipos de componentes ou nodos (GISOLFI, 2001), conforme ilustrado na Figura 3:

-*Service Provider*: é o componente que contém a lógica de funcionamento do serviço. É responsável por publicar as informações acerca do serviço prestado para que o mesmo possa ser descoberto e acessado por um cliente.

-*Service Broker*: atua como um registro de serviços, no qual o provedor registra o serviço fornecido e através do qual os clientes realizam a descoberta dos serviços.

-*Service Requester*: é a aplicação que faz uso dos serviços. Obtém as informações para acesso a um serviço e realiza solicitações ao mesmo. Pode ser também um outro serviço.

2.2. Web Services

Podem ser compreendidos como aplicações que podem ser acessadas através de URIs e cujas interfaces de comunicação podem ser definidas, descritas e descobertas (AFONSO et al.,

2004). Podem ser vistos como componentes que podem ser integrados em aplicações distribuídas complexas.

São aplicações autocontidas, que podem ser publicadas, localizadas e acessadas através de uma rede, normalmente a Web (IBM, 2000). Esse modelo é uma evolução do conceito de orientação a objetos, um paradigma conhecido nas linguagens de programação. A ideia é realizar uma separação lógica dos componentes, permitindo o encapsulamento da lógica da implementação e fornecendo APIs para que os serviços sejam acessados por clientes (IBM, 2000).

Segundo a W3C, os Web Services proveem um padrão para interoperabilidade entre aplicações, que utilizam diferentes plataformas, frameworks ou linguagens. Uma grande vantagem do uso de Web Services é permitir a interoperabilidade de aplicações heterogêneas. Isso significa que, através dos Web Services, é possível a comunicação de aplicações desenvolvidas em plataformas incompatíveis entre si. Por exemplo, uma aplicação desenvolvida na linguagem de programação C++ pode fazer uso dos serviços de um Web Service desenvolvido sobre a plataforma JAVA. Devido a esse fator, os Web Services são amplamente utilizados na integração de sistemas.

Em aplicações de grande porte, pode-se utilizar um conjunto de Web Services, de forma integrada, com baixo acoplamento, para realizar operações complexas (ORACLE, 2013).

Os padrões mais utilizados para construções de Web Services são o SOAP e o REST (ORACLE, 2013). Os Web Services SOAP são frequentemente chamados de ‘*Big Web Services*’, principalmente devido à utilização do formato XML para troca de mensagens, o que exige um processamento maior quando comparado ao formato JSON, comumente utilizado pelo padrão REST.

2.2.1.1.SOAP

SOAP é o acrônimo para *Simple Object Access Protocol*, porém o nome original entrou em desuso a partir da versão 1.2 (2007). O padrão ou protocolo SOAP foi definido pela W3C em 2001 e utiliza o formato XML para troca de mensagens.

Uma das principais características desse padrão é a utilização de um documento para publicação das informações de um serviço, o documento *WSDL (Web Service Description Language)*. Esse

documento contém informações descritas na linguagem XML acerca das operações e do formato dos dados de um serviço. Esse documento também pode ser interpretado por máquinas para extração automática das informações acerca do uso do serviço. O Quadro 1 apresenta o conteúdo de um documento *WSDL*.

O campo *types* contém informações acerca dos tipos de dados utilizados no serviço. As informações sobre os dados são descritas

```
<definitions>

<types>
--CONTEUDO--
</types>

<message>
--CONTEUDO--
</message>

<portType>
--CONTEUDO--
</portType>

<binding>
--CONTEUDO--
</binding>

</definitions>
```

Quadro 1 - Exemplo de envelope SOAP de uma requisição.

no formato XML Schema Definition (XSD). Esse campo pode ser omitido e os tipos de dados podem ser referenciados em arquivos XSD, que devem ser declarados em uma *tag* 'import'.

O campo *message* define os parâmetros esperados e quais os tipos retornados para cada uma das mensagens do serviço. As

mensagens são uma interação entre o serviço e o cliente, podendo ser uma mensagem de requisição ou de resposta. Esse campo deve referenciar os tipos definidos no campo *types* ou em arquivos XSD.

O elemento *portType* define as operações e as mensagens que envolvem uma operação. Refere-se ao nome da operação, o tipo de parâmetro esperado e o tipo de informação retornada pela operação.

O campo *binding* define informações sobre o formato dos dados e o protocolo utilizado para cada operação do serviço.

As mensagens trocadas entre o servidor e o cliente que realiza a requisição são codificadas em um envelope. O envelope encapsula as informações e contém algumas propriedades típicas de Web Services SOAP. No Quadro 2 é apresentado um exemplo de envelope SOAP para uma mensagem de requisição e no Quadro 3 um exemplo de resposta.

- Requisição:

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-
envelope"
soap:encodingStyle="http://www.w3.org/2001/12/s
oap-encoding">
<soap:Body xmlns:m="http://example.com/operacaoe
sAeroporto">
  <m:consultaVoo>
    <StatusVoo>
      <origem>GRU</origem>
      <destino>MCO</destino>
      <previsto>2015-08-
15T23:51:00.000Z</previsto>
      <confirmado>2015-08-
15T00:12:00.000Z</confirmado>
    </StatusVoo>
  </m:consultaVoo>
</soap:Body>
</soap:Envelope>
```

Quadro 2 - Exemplo de envelope SOAP de uma requisição.

- Resposta:

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding
">
<soap:Body xmlns:m="http://example.com/operacoesAero
porto">
  <m:consultaVoo>
    <ConsultaVooRequest>
      <companhia>AA</companhia>
      <voo>3522</voo>
    </ConsultaVooRequest>
  </m:consultaVoo>
</soap:Body>
</soap:Envelope>
```

Quadro 3 - Exemplo de envelope SOAP de uma resposta.

2.2.2. REST (*Representational State Transfer*)

REST é um modelo arquitetural para construção de Web Services cujo foco principal são recursos (FIELDING, 2000). O modelo preconiza a utilização do protocolo HTTP através de uma infraestrutura leve e a um baixo custo. Tais características justificam a sua grande adoção como padrão (ORACLE, 2013).

O fato de operar diretamente sobre o protocolo HTTP (ao contrário dos Web Services SOAP, que utilizam o protocolo SOAP) torna a adoção desse modelo como padrão de integração amplamente difundida. Algumas características colocam o padrão REST como principal padrão para integração de sistemas (FIELDING, 2000; ORACLE, 2013):

-Modelo cliente-servidor: A separação de responsabilidades é o princípio do modelo. O isolamento entre a implementação presente no servidor e a aplicação cliente permite que ambos os componentes evoluam de forma independente.

-*Stateless*: A comunicação não deve manter um estado entre o cliente e o servidor, ou seja, cada troca de mensagens deve ser independente. Cada mensagem trocada entre o cliente e o servidor deve conter toda informação necessária ao processamento da requisição. Essa propriedade permite o destaque de outras 3 propriedades: visibilidade, confiança e escalabilidade. A visibilidade refere-se ao fato de que é possível rastrear a origem de uma requisição de forma atômica, enquanto nas comunicações com manutenção do estado (*stateful*) a rastreabilidade das requisições depende de múltiplas requisições intermediárias. A confiança é alcançada pois o servidor pode recuperar-se facilmente de falhas, uma vez que as operações são únicas, e assim uma resposta depende unicamente da sua requisição. Por fim, a escalabilidade é alcançada uma vez que o sistema consegue liberar os recursos mais rapidamente, visto que não há a necessidade de armazenar informações intermediárias para processar requisições posteriores.

-*Cache*: O mecanismo de *cache* permite otimizar o uso dos recursos de rede. Diferentes recursos podem ou não utilizar *cache*. Para os recursos armazenados em *cache* pode-se eliminar a necessidade de interações via rede para obtenção do recurso, aumentando a eficiência e a escalabilidade e reduzindo o tempo de resposta das requisições.

-Interface Comum: Um grande diferencial da arquitetura REST é a utilização de uma interface comum entre os componentes. Sendo assim, a visão global do sistema é simplificada (menor complexidade dos componentes) e a visibilidade dos recursos é melhor (através da interface comum). Para alcançar a propriedade da interface comum deve-se seguir 4 princípios: identificação dos recursos, manipulação dos recursos pela sua representação, mensagens auto descritivas e utilização de hipermídia como núcleo da aplicação. A interface comum é uma das propriedades que permitem o desacoplamento entre o cliente e o servidor.

-Baixo consumo de recursos de rede: O fato de o modelo utilizar mensagens leves, diretamente sobre o protocolo HTTP,

permite a redução no consumo dos recursos da rede. Aliado a isso está o mecanismo de *cache*, que permite reutilizar informações previamente processadas. Essa propriedade é particularmente útil em um cenário de computação móvel, onde a redução no consumo dos recursos permite um melhor aproveitamento dos componentes do dispositivo, que são limitados devido a restrições de energia.

-Sistema multicamadas: A arquitetura REST é particularmente útil para desenvolvimento de sistemas em várias camadas devido ao seu direcionamento à obtenção de recursos. Tal fator permite decompor uma aplicação em camadas, mantendo a comunicação mínima entre as camadas através de serviços. Os serviços podem integrar aplicações legadas ou isolar clientes de alterações no servidor.

2.3. Qualidade de Serviço (QoS)

Qualidade de serviço pode ser entendida como um conjunto de parâmetros que são medidos com o intuito de verificar se os requisitos exigidos para um cliente são atendidos pelo provedor do serviço (ITU-T, 1994). Inicialmente, o termo foi vinculado a telecomunicações e redes de computadores. Porém, quando utilizado no contexto de aplicações computacionais, sobretudo aplicações orientadas a serviços, os parâmetros de QoS permitem associar à aplicação um grau de confiança para os clientes acerca dos serviços fornecidos.

No caso dos Web Services, a qualidade de serviço está associada às seguintes propriedades não-funcionais (MANI-NAGARAJAN, 2002):

-Disponibilidade: Refere-se a quando um serviço está presente e disponível para uso imediato. A quantificação desse parâmetro se dá em porcentagem e representa a probabilidade de um serviço estar disponível para uso.

-Acessibilidade: Representa o grau em que um serviço é capaz de prover uma resposta para uma requisição. É expresso na forma de probabilidade, medindo a taxa de sucesso ou a chance de sucesso de uma requisição ao serviço em um determinado momento. Difere

do parâmetro de disponibilidade, uma vez que um serviço pode estar disponível, mas não respondendo a requisições.

-Integridade: Refere-se à propriedade de um serviço em manter uma resposta correta ao longo de uma transação. Transações são uma sequência de interações que podem ser identificadas como uma unidade (BERNSTEIN-NEWCOMER, 2009). Cada interação deve completar com sucesso para que a transação possa finalizar corretamente.

-Desempenho: É uma medida que avalia o serviço em termos de capacidade e latência. Capacidade refere-se ao número de requisições processadas em um período de tempo, e latência ao tempo gasto entre o envio de uma requisição e o recebimento da resposta. Serviços com um bom desempenho apresentam uma baixa latência e uma alta capacidade.

-Confiança: Diz respeito à propriedade de um serviço em se manter em funcionamento e com qualidade. Uma medida de confiança pode ser o número de falhas em um mês ou em um ano.

-Regularidade: Refere-se à propriedade de um serviço em manter-se em conformidade com as leis, regras, padrões e níveis de serviço acordados. Em geral, Web Services seguem padrões e regras que devem ser respeitados ao longo do seu funcionamento, sob pena de prejudicar o funcionamento de aplicações cliente que esperam o serviço funcionando dentro de parâmetros pré-estabelecidos.

-Segurança: Refere-se à confidencialidade e não-repúdio através da autenticação das partes envolvidas na comunicação, encriptação das mensagens ou controle de acesso. A segurança é um aspecto de suma importância para os Web Services, uma vez que toda a comunicação ocorre diretamente sobre a internet, onde as mensagens trocadas podem ser interceptadas ou alteradas. O parâmetro que diz respeito à segurança pode ser controlado de acordo com as exigências da aplicação solicitante, permitindo diferentes níveis de controle.

2.4. Considerações Finais do Capítulo

Esse capítulo teve como objetivo apresentar os principais tópicos relacionados aos serviços web. Foi apresentada a arquitetura orientada a serviços e os modelos de serviço REST e SOAP. Além

disso, foram apresentados os conceitos acerca da qualidade de serviço.

No próximo capítulo serão apresentados os principais tópicos sobre a arquitetura de microservices.

3. MICROSERVICES

O termo *microservices* refere-se a um modelo de desenvolvimento de aplicações (NAMIOT-SNEPS, 2014). *Microservices* podem ser compreendidos como aplicações de pequeno porte, desenvolvidas como serviços e que operam em conjunto (NEWMAN, 2015). São comparados a componentes, ou seja, podem ser evoluídos ou substituídos de forma independente. Cada serviço executa seu próprio processo, deve ser leve e independente e deve possuir uma interface bem definida, diretamente sobre o protocolo HTTP, para comunicação com outros serviços. (NAMIOT-SNEPS, 2014).

Nesse modelo de arquitetura as fronteiras de um serviço podem ser mapeadas diretamente para as fronteiras das regras de negócio. É fácil perceber onde se encontra um determinado código responsável pela execução de uma funcionalidade (NEWMAN, 2015). Indo mais além, é fácil perceber que o impacto de uma mudança em uma funcionalidade é restrito às fronteiras do serviço onde ela reside.

Esse modelo de arquitetura para desenvolvimento de aplicações vem sendo adotado com frequência por grandes empresas, a exemplo da GILT que é uma empresa de comércio eletrônico para venda de artigos de vestuário e acessórios. A empresa trabalha com promoções repentinas e isso faz com que o acesso ao sistema tenha picos elevados alternando com momentos de relativa ociosidade. O sistema para vendas on-line da GILT foi originalmente concebido como uma aplicação monolítica. Entretanto, com o crescente aumento no número de acessos e operações, o modelo originalmente adotado foi tornando-se incapaz de suprir os elevados picos (BRYANT, 2015).

Com vistas a resolver o problema, decidiu-se evoluir a aplicação para uma arquitetura de *microservices*. No ano de 2015, a aplicação atingiu o número de 156 *microservices*. Algumas das vantagens obtidas com a nova arquitetura foram (BRYANT, 2015):

- Menos dependência entre as equipes de desenvolvimento, resultando em versões mais rápidas dos serviços.
- Permitir o desenvolvimento de inovações (novos *frameworks*, linguagens, etc.) nos serviços.
- Código dispensável, ou seja, pode ser facilmente substituído ou refatorado.
- Degradação segura de um serviço, alcançada através da independência na operação dos serviços, de modo que caso um serviço apresente problemas de desempenho, não prejudicará os demais.

3.1. Dividir para Conquistar

A máxima que é utilizada em diversas áreas de conhecimento refere-se ao processo de dividir um problema em problemas menores, de modo que se resolvendo os problemas menores chegue-se à solução do problema como um todo. Na área de Ciência da Computação, o termo foi empregado para descrever uma técnica de algoritmo recursivo para solução de problemas computacionais (KARATSUBA-OFMAN, 1962).

O termo pode ser utilizado de forma análoga ao conceito de *microservices*. Esse modelo é essencialmente a aplicação da metodologia para a fragmentação de uma aplicação em partes (serviços) menores. O conjunto de serviços, atuando de forma integrada, reflete o comportamento esperado para a aplicação como um todo (FOWLER-LEWIS, 2014).

Quando bem estruturados, os serviços fragmentados de uma aplicação monolítica vão executar apenas uma tarefa bem-definida (NEWMAN, 2015). Através dessa abordagem, é possível evitar que um serviço cresça deliberadamente, como acontece com as aplicações monolíticas.

A independência entre os serviços permite que novas versões possam ser lançadas com uma frequência maior, contendo um número menor de novas funcionalidades e evitando o acúmulo de funcionalidades que acabam por ocasionar uma dificuldade na validação de uma versão (APPLE, 2014). Ou seja, a integração contínua está naturalmente presente no ciclo de desenvolvimento dos *microservices*.

3.2. Independência Tecnológica

Uma bandeira levantada pelos defensores da arquitetura de microservices é o fato de que os serviços podem/devem ser desenvolvidos em tecnologias diferentes, ou seja, para cada serviço deve-se utilizar a tecnologia mais adequada (HOFF, 2014).

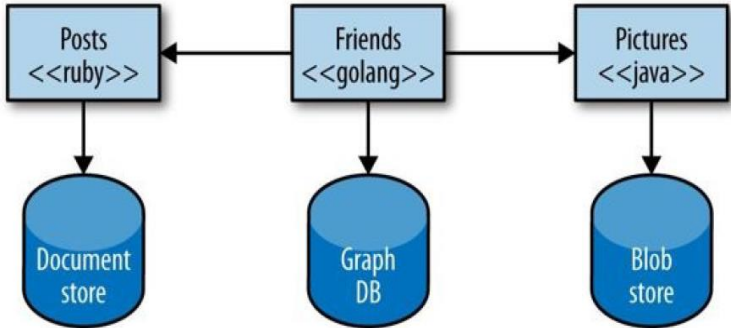


Figura 4 - Microservices com Diferentes Tecnologias (NEWMAN, 2015).

A Figura 4 mostra um conjunto de microservices que se comunicam entre si e utilizam diferentes tecnologias. A opção por uma tecnologia em especial pode considerar diversos fatores: experiência da equipe, desempenho, facilidade de manutenção, facilidade para integração com uma base de dados, entre outros. O importante é utilizar a tecnologia que melhor se aplica para as funcionalidades do serviço.

Há, contudo, uma ressalva. Para que os serviços possam se comunicar, é necessária uma interface comum, ou seja, um padrão fixo para que a troca de informações possa ocorrer. É recomendado que seja utilizado um padrão capaz de operar diretamente sobre o protocolo HTTP, como os Web Services REST (FOWLER-LEWIS, 2014). Uma vez estabelecida a interface comum, um serviço pode evoluir internamente sem que haja qualquer prejuízo aos serviços que com ele se comunicam.

As tecnologias para desenvolvimento de aplicações evoluem rapidamente. Sempre há um risco associado à adoção de novas tecnologias em uma aplicação (NEWMAN, 2014). Porém, no

modelo de microservices, esse risco é reduzido consideravelmente, pois os serviços são simples e podem ser facilmente substituídos na sua totalidade. O risco pode ser reduzido ainda mais caso a tecnologia a ser adotada seja previamente testada em um serviço de menor complexidade, sabendo que o impacto da adoção será baixo (NEWMAN, 2014).

3.3. Resiliência

É um conceito que remete à capacidade de uma aplicação de se recuperar frente a situações imprevistas e que impeçam o seu funcionamento regular (BONÉR et al, 2014). A resiliência é alcançada através de replicação, contingenciamento, isolamento e delegação. As falhas podem ser isoladas em componentes e isso garante que as partes do sistema que apresentam o comportamento anormal possam se recuperar sem comprometer o sistema como um todo (BONÉR et al, 2014). Por outro lado, em uma aplicação monolítica, uma falha em um dos componentes pode prejudicar o funcionamento de todo o sistema.

O problema é que falhas são frequentes. A melhor abordagem é assumir que, em algum momento, elas vão ocorrer. A resiliência pode ser alcançada com o foco em alguns aspectos (NEWMAN, 2014):

- Tempo de resposta: O tempo para processar uma determinada requisição deve ser constantemente medido e avaliado. Devido à forma como são construídas as redes de comunicação, um número excessivo de conexões simultâneas pode impactar diretamente no tempo de resposta. É importante conhecer a capacidade de processamento da aplicação bem como da infraestrutura sobre a qual a mesma está executando. Dessa forma, através de um monitoramento efetivo, podem ser feitas alterações na infraestrutura para fragmentar as conexões sem prejudicar o tempo de resposta.
- Disponibilidade: Um serviço irá, em algum momento, enfrentar um período de indisponibilidade. É importante garantir que o período de indisponibilidade seja aceitável sob o ponto de vista de um cliente que está usando o serviço. A medição dos períodos de indisponibilidade ajuda a analisar um histórico e concluir se as perdas são aceitáveis.

- **Durabilidade dos Dados:** Refere-se à aceitabilidade na perda dos dados e o período de retenção dos mesmos. Diferentes tipos de dados possuem restrições únicas. Por exemplo, os logs de acesso de um usuário podem ser mantidos por um período de um ano, enquanto registros de transações financeiras podem necessitar um período de retenção de muitos anos.

3.4. Implantação (*Deploy*)

Refere-se ao processo pelo qual uma aplicação é agregada em um pacote contendo todos os arquivos necessário à sua execução e posterior execução do pacote em um servidor.

A fragmentação de uma aplicação em um conjunto de serviços torna a tarefa de implantação individual mais rápida e segura. Tal isolamento é essencial, principalmente, para que novas versões dos serviços possam ser integradas rapidamente e com baixo impacto na aplicação como um todo (NAMIOT-SNEPS, 2014). Esse é um dos pontos negativos das aplicações monolíticas. Para cada nova funcionalidade, a aplicação inteira deve ser substituída.

Há, contudo, controvérsias. Se considerarmos a implantação da aplicação como um todo, para que vários serviços dependentes sejam implantados simultaneamente é necessário que a implantação ocorra de forma sincronizada (HOOF, 2015). A tarefa é ainda mais árdua quando os serviços são compostos por diferentes linguagens e tecnologias. O gerenciamento da implantação de dezenas de serviços de forma manual torna-se impraticável. Nesse sentido, existem ferramentas de automatização do processo de implantação, como Puppet⁴ e Capistrano⁵ (DVORKIN, 2014).

A implantação é um dos pontos em que as aplicações monolíticas podem levar uma vantagem quando comparadas à arquitetura de microservices. A menos, é claro, que a aplicação monolítica tenha uma complexidade muito grande (NAMIOT-SNEPS, 2014). Essa

⁴ <https://puppetlabs.com/>

⁵ <http://capistranorb.com/>

complexidade pode ocasionar grandes problemas de integração de novos componentes. Torna-se difícil controlar o impacto das alterações e, caso haja um problema, a rastreabilidade do mesmo torna-se complicada (NEWMAN, 2014). Esse problema tende a crescer proporcionalmente com a adição de novas funcionalidades. De forma oposta, problemas durante a implantação de um serviço na arquitetura de microservices podem ser facilmente superados através de um isolamento e *rollback* do serviço que apresenta problema (NEWMAN, 2014).

3.4.1. Ferramenta *Docker*

O *Docker* é uma plataforma para execução e portabilidade de aplicações distribuídas. Permite a criação de pilhas de softwares que são utilizadas para execução de aplicações. Através da plataforma é possível isolar a aplicação e suas dependências da infraestrutura. Caso seja necessário alterar a infraestrutura sobre a qual a aplicação está executando, basta portar o *docker* e automaticamente todas as dependências e a aplicação são portados de forma transparente e como uma unidade.

Para prover o ambiente de gerenciamento das aplicações o *Docker* utiliza *contêineres*. Os *contêineres* são o ambiente de execução das aplicações. O *Docker* gerencia as aplicações e suas dependências através de imagens (*docker images*). Em um repositório próprio milhares de imagens já foram criadas e estão disponíveis para uso, por exemplo, uma imagem com servidor de aplicação *wildfly* em um sistema centos com *jdk8*. É possível também criar novas imagens customizadas e adicioná-las a um repositório público ou privado.

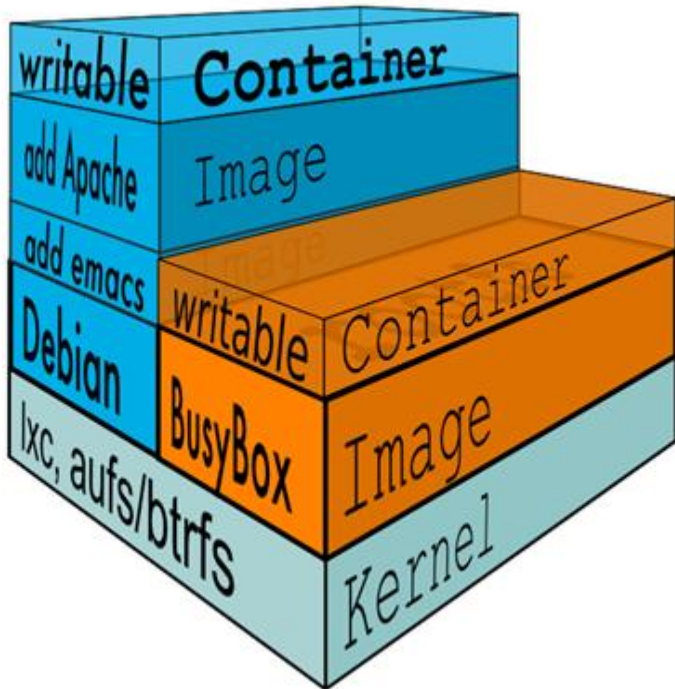


Figura 5 – *Docker*: contêiner e imagens (DOCKER, 2015).

A Figura 5 mostra a interação entre os contêineres e as imagens. Os containers são responsáveis por gerenciar a execução das imagens. Em uma única máquina diversos containers podem ser executados de forma isolada, porém compartilhando os recursos do sistema. Os containers utilizam diversos padrões para execução e gerenciamento dos recursos o que permite que sejam executados em diversas distribuições Linux e Windows.

A criação de novas imagens parte de uma imagem base, geralmente o sistema operacional. Ao criar uma nova imagem o *Docker* armazena apenas a diferença entre a imagem base e a nova imagem. Quando a nova imagem for executada em um contêiner a plataforma coloca a imagem em uma camada acima da imagem base utilizando um sistema de arquivos em camada (comumente o

AUFS). Por fim, o AUFS realiza um *merge* das camadas. Mais camadas (novas imagens) podem ser adicionadas indefinidamente.

Os conceitos e o modo de funcionamento do *Docker* muito se assemelham com as máquinas virtuais, porém há algumas diferenças:

-Compartilhamento eficiente de recursos: O *Docker* permite que os recursos do sistema sejam compartilhados visando uma melhor eficiência. Através de mecanismos para isolamento do sistema de arquivos é possível criar áreas que são compartilhadas e outras que não são.

-Portabilidade: O *Docker* permite que toda a aplicação e suas dependências sejam facilmente realocadas em outra infraestrutura. Mantendo o ambiente de execução intacto, a nova infraestrutura não afeta o funcionamento da aplicação. No caso das máquinas virtuais a aplicação fica vinculada à infraestrutura, desse modo, é necessário instalar novamente a aplicação na nova máquina virtual.

-Testes e validação: Com o *Docker* é possível criar ambientes de execução que conseguem manter a consistência mesmo em diferentes níveis da instituição: por exemplo, o ambiente de desenvolvimento deve seguir fielmente o ambiente de produção. Dessa forma, os testes realizados no ambiente de desenvolvimento são validados para o ambiente de produção. Também é fácil realizar testes de desempenho em diferentes infraestruturas (com a máquina virtual diversas instancias devem ser criadas e os dados são replicados). *Docker* torna fácil a criação de novas configurações para testar a aplicação.

-Isolamento: O *Docker* permite que erros possam ser identificados de forma rápida e ações corretivas possam ser tomadas para corrigir o problema. O isolamento na execução do container permite realizar um *rollback* apenas na porção onde o problema foi identificado, tornando a ação mais eficiente e menos destrutiva do que o modelo tradicional.

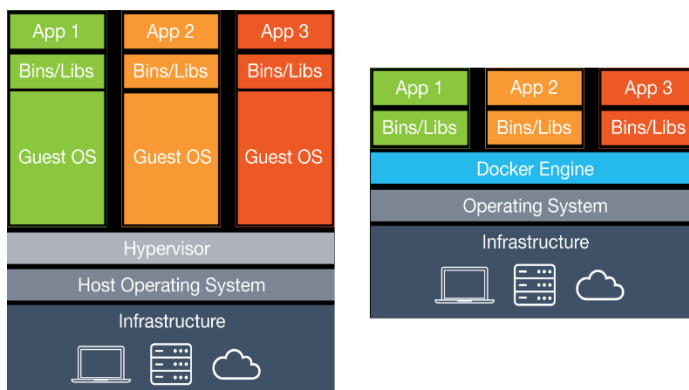


Figura 6 – *Docker* e Máquina Virtual (DOCKER, 2015).

A Figura 6 mostra como o *Docker* atua para isolar a execução de três aplicações com diferentes configurações, executando sobre o mesmo sistema operacional (*kernel*). No caso das máquinas virtuais é necessário um novo sistema operacional para cada máquina.

3.5. Limitações

A arquitetura de microservices possui também limitações ou desafios que acabam por ocasionar dificuldades na sua construção, manutenção, instalação e monitoramento. Muitos desses problemas podem ser superados através da utilização de ferramentas, padrões de desenvolvimento e boas práticas. A seguir são enumerados alguns dos problemas da arquitetura e suas possíveis soluções (HOFF, 2014; DVORKIN, 2015):

- 1) **Custo Operacional:** É fato que quando comparados a uma única aplicação monolítica, os microservices representam um grande custo extra para instalação das aplicações nos servidores. O problema cresce potencialmente com o aumento no número de serviços. Além disso, é frequente a necessidade de orquestração, um processo complexo e que demanda um conhecimento prévio de ferramentas específicas para essa tarefa. Outro fator relevante é o próprio provisionamento de hardware, isto é, o gerenciamento da

infraestrutura para dezenas de serviços. Solução: A solução para esse problema é a utilização de ferramentas de automatização. A ideia é automatizar o processo de instalação das aplicações e do provisionamento dos recursos do hardware. Podem ser utilizadas ferramentas de auto escalonamento, que proveem alocação dinâmica e um melhor aproveitamento de recursos de hardware.

- 2) Monitoramento: O monitoramento de aplicações monolíticas demanda a concentração de recursos em uma única fonte, o servidor onde a aplicação está instalada. A detecção de anomalias no funcionamento da aplicação, além da detecção de degradação de desempenho, pode ser imediata. O problema é que para dezenas de serviços a tarefa torna-se bastante complexa, principalmente devido à quantidade de serviços e também ao uso de diferentes tecnologias no conjunto de serviços. Em muitas ocasiões, os serviços comunicam-se entre si, e identificar a fonte da degradação de desempenho torna-se difícil. É comum a utilização dos mesmos recursos de hardware para múltiplos serviços. Nesse contexto, o monitoramento dos recursos de hardware não reflete uma única aplicação, mas sim um conjunto de serviços. Solução: São muitas as ferramentas para monitoramento de aplicações. O problema é que essas ferramentas são ideais para monitoramento de uma aplicação. Não há uma ferramenta capaz de agregar o monitoramento de dezenas de serviços. Espera-se que o presente trabalho possa suprir tal necessidade agregando o monitoramento e a rastreabilidade em uma única solução. Para detectar anomalias no funcionamento dos serviços ou durante a integração dos serviços é recomendada a utilização do monitoramento sintético ou ativo, que visa simular requisições para os serviços a fim de detectar quando ocorrem erros nas respostas esperadas.
- 3) Maior Expertise: Enquanto no desenvolvimento de aplicações monolíticas são utilizados um único servidor ou um cluster com poucos servidores, no caso dos microservices há um aumento na complexidade operacional e de desenvolvimento. O desenvolvimento demanda um maior conhecimento de ferramentas em múltiplos campos de conhecimento. Por exemplo, enquanto as aplicações monolíticas fazem uso de um único banco de dados, os microservices podem utilizar diversos bancos do tipo NoSQL. Nesse sentido, a equipe de desenvolvimento deve dominar as tecnologias para instalar, otimizar, executar e manter seus serviços dentro dos requisitos exigidos. A equipe de suporte precisa ser

extremamente proativa para gerenciar dinamicamente o conjunto de serviços e apoiar a equipe de desenvolvimento na construção dos mesmos. A complexidade no gerenciamento dos recursos é substancialmente maior quando comparada ao gerenciamento de aplicações monolíticas. Solução: Para esse problema não há solução definitiva. A expertise da equipe depende de conhecimento prévio ou experiência. Nesse sentido, é essencial que haja pessoas com amplo conhecimento da arquitetura de microservices, em especial no que tange os problemas que podem surgir. O conhecimento prévio permite uma atuação proativa no sentido de minimizar ou conter situações que prejudiquem o desenvolvimento ou manutenção dos serviços.

- 4) Rastreabilidade de Requisições: O problema é manter um registro das requisições que dependem de um fluxo através de diversos serviços. A rastreabilidade de uma requisição através dos serviços permite identificar o ponto onde ocorre um problema no processamento da mesma. Além disso, pode ser necessário manter um registro isolado acerca da utilização dos serviços de forma individual. Solução: Para solucionar o problema da rastreabilidade pode ser utilizada a técnica do ID de correlação. Essa técnica é utilizada em sistemas distribuídos e consiste em gerar um valor durante o processamento da requisição no primeiro serviço e em seguida propagar esse valor durante o processamento nos demais serviços.
- 5) Esforço Duplicado: Refere-se ao fato de que, em muitas ocasiões, diferentes serviços dependem das mesmas funções. O problema é que os serviços utilizam diferentes tecnologias e muitas são incompatíveis. Sendo assim, não é possível compartilhar uma biblioteca entre os serviços. Solução: O problema pode ser resolvido através da criação de um serviço que provê a função que deve ser compartilhada. Porém, essa solução acaba por gerar um certo grau de acoplamento entre os diferentes serviços. A centralização da lógica em um único serviço também dificulta a evolução e manutenção dos demais serviços. Outra opção para solucionar o problema é duplicar a função entre os serviços. Essa solução, contudo, vai de encontro às melhores práticas no

desenvolvimento de software, pois uma alteração na função acaba por gerar um esforço de manutenção nos serviços que a utilizam.

- 6) Comportamento Assíncrono: microservices são mais propensos a utilizar funções assíncronas para prover suas funcionalidades. O objetivo é melhor utilizar os recursos do sistema e otimizar o tempo de resposta para uma requisição. No entanto, é complicado controlar o fluxo de informações. As execuções ocorrem fora de ordem e podem ter durações diferentes. Solução: Para esse problema, pode ser utilizado um ID de correlação para prover a rastreabilidade das requisições. Para as operações assíncronas muitos problemas podem ser identificados através do isolamento e testes dos serviços de forma separada.
- 7) Testes: A evolução individual e a próprio passo de cada serviço acaba por gerar um problema na manutenção e evolução dos cenários de teste. Diferentes serviços necessitam de diferentes ambientes de execução, o que aumenta a complexidade dos testes. Aliado a isso está o fato de que devem ser testados fluxos que dependem da execução de diferentes serviços. Cada serviço pode ser testado de forma isolada, porém problemas podem emergir repentinamente após a interação entre os serviços. Solução: Utilizar a técnica *canary release*. Essa técnica tem o objetivo de reduzir o risco de introduzir um novo componente em um ambiente em execução. A ideia é liberar de forma gradual a funcionalidade no ambiente, dessa forma, apenas um conjunto de usuários irá acessar os novos componentes e, após a validação, as novas funcionalidades são estendidas a todos os usuários (SATO, 2014).

3.6. Considerações Finais do Capítulo

Esse capítulo teve como objetivo apresentar os principais tópicos relacionados aos microservices. Foi apresentada uma definição acerca da arquitetura de microservices, a sua organização e a diferença entre os microservices e a estrutura tradicional utilizada nas aplicações monolíticas. Além disso, foram apresentadas as principais vantagens e desvantagens no uso dessa arquitetura.

No capítulo seguinte é apresentada uma revisão sobre testes de software, testes em serviços e também as principais ferramentas utilizadas para a execução de testes em serviços web.

4. TESTE DE SOFTWARE

Teste de software é o processo de execução de um programa com a intenção de encontrar erros (MYERSON, 2011). O teste de software é dividido em dois tipos principais: caixa branca e caixa preta. O teste do tipo caixa preta consiste no não conhecimento acerca do funcionamento interno do software a ser testado, ou seja, o teste é executado considerando-se apenas os valores de entrada e saída. Por outro lado, os testes do tipo caixa branca consideram o funcionamento interno do software, fazendo com que cada instrução interna seja executada pelo menos uma vez.

No campo de testes de software existem diferentes estágios ou níveis de teste (THAKARE, 2012; SATHAWORNWICHIT, 2014):

- Teste Unitário: O objetivo dessa fase de testes é testar cada componente do software ou cada unidade individualmente, sem considerar outras partes da aplicação.
- Teste de Integração: Nesse estágio o foco dos testes é na verificação dos componentes após a sua integração com outros componentes. Para a execução dessa fase de teste, os componentes da aplicação são conectados e montados em um único artefato.
- Teste de Sistema: Nessa fase de testes são executados os testes fim a fim. O objetivo desse estágio é a verificação das funcionalidades da aplicação frente aos requisitos que foram levantados na fase de análise. Também é verificado se todos os requisitos estão sendo atendidos pela aplicação.
- Teste de Aceitação: Nesse nível de teste os usuários da aplicação executam os testes diretamente na aplicação para verificar se a solução entregue atende as suas necessidades.

4.1.1. Testes de Desempenho

Os testes de desempenho são classificados como do tipo caixa preta e são executados na fase de teste de sistema. Considerando uma aplicação composta por microservices, o teste de desempenho é também chamado de teste fim a fim, onde as requisições são

enviadas ao serviço a partir de uma aplicação cliente através da rede de comunicação. Nesse tipo de aplicação não há uma interface gráfica, e os testes são executados considerando as operações que são fornecidas pelo microservice.

Considerando os testes de desempenho, o propósito do teste é verificar que o software cumpre com o conjunto de requisitos não funcionais como vazão, tempo de resposta e disponibilidade. O termo teste de desempenho é comumente utilizado para referir-se a um conjunto de diferentes tipos de testes de desempenho (THAKARE, 2012; SATHAWORNWICHIT, 2014) como teste de stress, carga e capacidade. Sendo assim, para o contexto deste trabalho foi utilizado o termo teste de desempenho na sua forma genérica, uma vez que a arquitetura proposta pode ser utilizada em qualquer um dos tipos de teste.

4.1.2. Testes de Serviços

Nas aplicações que seguem a arquitetura SOA, os testes de desempenho são realizados tendo como alvo as operações do serviço. Existem diversas ferramentas que podem ser utilizadas para execução de teste de desempenho em serviços SOA, dentre as quais podemos citar o JMeter e o SoapUI, que serão descritos ao final deste capítulo.

Previamente à execução dos testes de desempenho, o usuário deve definir o caso de teste a ser utilizado. Os dados que irão compor o modelo dependem da operação a ser testada. No caso de serviços construídos com base no estilo arquitetural REST, uma operação que utiliza o método HTTP GET não possui conteúdo no corpo da requisição. Caso seja uma operação do tipo POST, o usuário pode incluir um conteúdo para ser enviado no corpo da requisição.

Os testes de desempenho de serviços normalmente requerem diversos passos que devem ser executados manualmente pelo usuário que executa o teste:

- Definição da operação a ser testada.
- Definição do caso de teste, pode incluir parâmetros de cabeçalho, o conteúdo da requisição e parâmetros da URI.
- Definição dos parâmetros de teste da aplicação cliente: número de threads, número de requisições, tempo de duração, entre outros.

Dentre as etapas para execução do teste de desempenho a etapa que pode requerer um maior esforço é a definição do caso de teste. Essa etapa exige do usuário um conhecimento acerca do serviço, dos parâmetros a serem enviados e dos campos que compõem os parâmetros no caso de uma requisição com o método Post.

Frente a essa dificuldade o foco do presente trabalho é em buscar uma alternativa para que esse modelo de requisição seja automaticamente obtido pela aplicação de teste. Com a especificação sendo fornecida pelo próprio serviço, o usuário que deseja testar a aplicação precisa apenas definir a operação a ser testada e os parâmetros de teste.

4.2. Ferramentas para Teste em Web Services

Nesta seção são descritas duas ferramentas voltadas para automatização de testes de Web Services, que também podem ser utilizadas no contexto de microservices.

4.2.1. Ferramenta *SoapUI*

Lançada originalmente em 2005, a ferramenta foi desenvolvida para auxiliar a execução de testes funcionais em Web Services SOAP. A ferramenta é livre e de código aberto. Algumas funcionalidades da versão 5.1 são: testes funcionais, testes de segurança, testes de carga, geração de relatórios, geração de modelos de requisições para REST e SOAP, entre outras.

Uma funcionalidade importante da ferramenta é a execução de testes de carga. Os testes de carga permitem avaliar um serviço através de diversos processos distribuídos simulando diferentes quantidades de clientes consumindo o serviço de forma simultânea. Também é possível integrar o SOAPUI a uma API permitindo distribuir os clientes de testes em máquinas virtuais e simulando um cenário mais próximo do real.

Há também uma versão comercial da ferramenta, que adiciona suporte a definição de scripts para validação das requisições e respostas, simulação de serviços, *debug* de testes, entre outros.

A *Figura 7* apresenta um exemplo de requisição e resposta utilizando a ferramenta SOAPUI. No exemplo apresentado o serviço é um Web Service SOAP. Os valores da requisição foram inseridos de forma manual. Uma vez estipulado o modelo de requisição, a ferramenta permite a execução dos testes de carga.

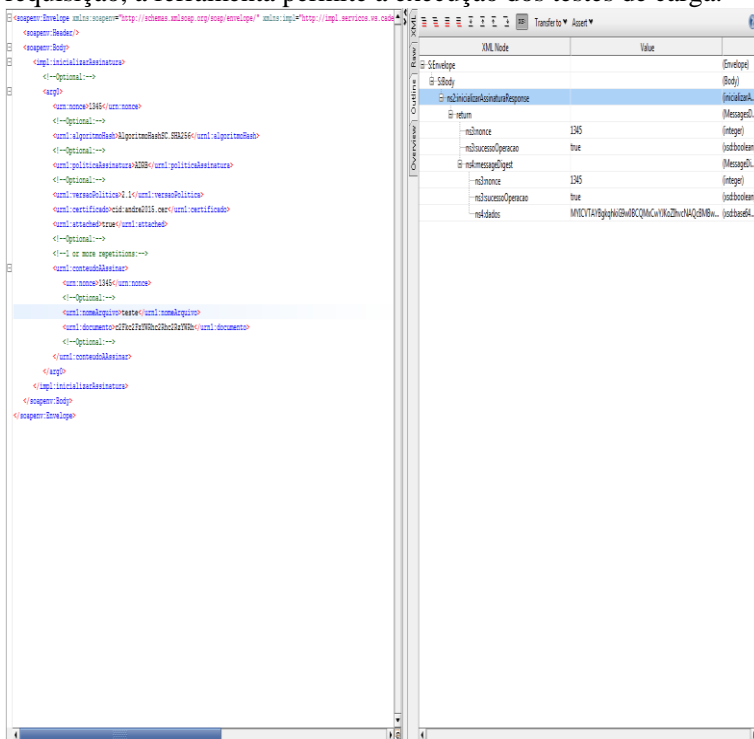


Figura 7 - Exemplo de Requisição e Resposta utilizando o SOAPUI

Os testes de carga na ferramenta SOAPUI permitem a extração de diversas métricas como: tempo de resposta máximo, mínimo e médio, número de requisições enviadas, quantidade de transações por segundo, quantidade de bytes por segundo, entre outras. A *Figura 8* mostra um exemplo de teste de carga composto por duas requisições.

The screenshot shows the SoapUI TestRunner interface. At the top, there are controls for 'Threads' (set to 50), 'Strategy' (Simple), and 'Test Delay' (0). A 'Results' table is displayed below, showing performance metrics for different test steps. The 'TestCase' row is highlighted in green.

Test Step	min	max	avg	std	err	tps	bytes	bytes	err	err
initialization	60	1134	668.24	303	4370	67.85	5305300	93803	0	0
TestRunner	18	2286	82.93	36	4870	87.85	14884300	207069	0	0
TestCase	18	13037	329.28	3209	4878	87.85	20380000	338273	0	0

Figura 8 - SOAPUI: Teste de Carga

As estatísticas são coletadas e apresentadas de forma individual, para cada uma das requisições. Ainda pode ser controlada a quantidade de *threads*, de modo a simular múltiplos clientes, e o tempo de execução do teste. A ferramenta ainda apresenta estratégias de teste que permitem simular diversos comportamentos em um cenário real:

- **Fixed Rate:** Especifica a quantidade de casos de teste que serão executados por segundo. A ferramenta vai alocar e variar o número de threads para manter o valor especificado.
- **Variance:** Varia o número de threads ao longo de um período para simular um fluxo variado de requisições.
- **Burst:** Realiza rajadas para impor ao serviço uma carga elevada e analisar a sua capacidade de recuperação.
- **Thread:** Permite que a quantidade de threads seja alterada ao longo de um período. Este tipo de teste serve, por exemplo, para identificar a quantidade ótima de clientes, ou seja, a quantidade para a qual o serviço fornece uma melhor taxa de transações por segundo (tps).

O SOAPUI é uma ferramenta robusta e permite a execução de uma grande quantidade de testes. Porém, a requisição a ser utilizada para consumir os serviços deve ser especificada de forma manual (apesar de a ferramenta gerar automaticamente o modelo, os valores são informados manualmente). Esse é um dos problemas

para os quais o presente trabalho busca uma solução. É prudente vincular a especificação dos testes ao serviço, pois, garante-se que ambos estarão sempre em consistência. Dessa forma, pode-se gerar um mecanismo automatizado capaz de executar os testes de carga com base nas especificações presentes em cada um dos serviços, sem que haja a necessidade de uma intervenção manual.

4.2.2. Ferramenta *JMeter*

O *JMeter* é uma ferramenta desenvolvida especificamente para testes de carga, que permite realizar os testes em aplicações desktop ou serviços. Seus recursos incluem também a geração de casos de testes com customização da validação dos resultados, além da geração de relatórios e gráficos.

Por sua natureza voltada a execução de testes de carga, o *JMeter* possui diversos recursos que permitem a definição de casos de teste bastante complexos. A Figura 9 apresenta a interface para definição de uma requisição Http utilizando o protocolo SOAP.

Figura 9 – *JMeter*: requisição a um serviço SOAP.

A Figura 10 mostra os parâmetros que são definidos previamente a execução do teste de carga, sendo: número de threads, o período

de espera entre o início de cada thread (*ramp-up*) e por fim a quantidade de vezes que o teste vai ser executado (*loop count*).

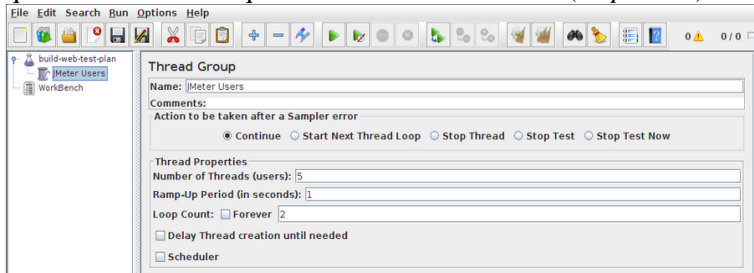


Figura 10 - *JMeter*: Definição dos Parâmetros de Teste.

Ao final do teste de carga é possível gerar um relatório (mostrado na Figura 11) que contém os resultados obtidos. Algumas das métricas coletadas são similares àquelas apresentadas pelo SOAPUI: tempo médio de resposta, tempo máximo e tempo mínimo e *throughput*, que é equivalente ao *tps* do SOAPUI.

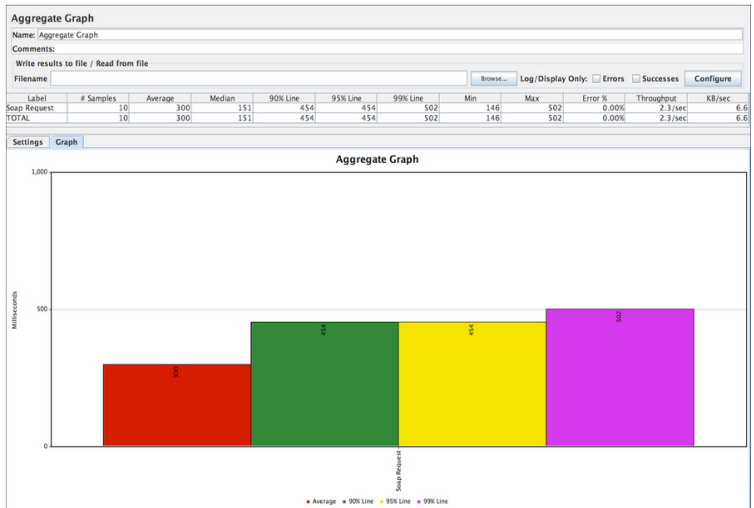


Figura 11 - *JMeter*: Relatório de Teste.

Apesar da vasta documentação, a ferramenta apresenta uma curva de aprendizado relativamente superior quando comparada ao

SOAPUI, principalmente, devido ao grande número de recursos que a mesma possui.

A ferramenta consegue prover diversos recursos para a execução de diferentes cenários de teste. Porém, carece do mesmo problema do SOAPUI: as requisições para a execução dos testes devem ser feitas manualmente. Ou seja, qualquer alteração no serviço implica em uma alteração na especificação do teste. Em um cenário contendo uma aplicação composta por um conjunto de microservices, isso significa que cada serviço deve ser especificado de forma manual, e, para cada alteração, uma nova especificação deve ser gerada.

No contexto do presente trabalho optamos por utilizar a ferramenta JMeter por ser uma ferramenta de código aberto, com uma documentação extensa quando comparada ao SOAPUI e também por ter um conhecimento prévio acerca do uso da ferramenta.

4.3. Considerações Finais do Capítulo

Neste capítulo foi apresentada uma revisão acerca dos testes de software, seus tipos, e também como são executados os testes em serviços web.

As ferramentas apresentadas neste capítulo são amplamente utilizadas para execução dos testes de desempenho em serviços. Essas ferramentas podem ser utilizadas em conjunto com a arquitetura proposta neste trabalho para execução dos testes de desempenho nos microservices.

No próximo capítulo será apresentado o estado arte com os trabalhos relacionados.

5. ESTADO DA ARTE

Este capítulo tem como objetivo apresentar o estado da arte em pesquisas e trabalhos que estão diretamente relacionadas com testes de software, com ênfase nos testes de desempenho de microservices ou de Web Services. As buscas foram estendidas para trabalhos relacionados a especificação de testes em Web Services devido à escassez de trabalhos relacionados a testes em microservices.

5.1. Trabalhos Correlatos

No levantamento do estado da arte realizado durante este trabalho, não foi possível encontrar na literatura trabalhos que tratem especificamente de testes de desempenho em microservices. Deste modo, foram analisados também os trabalhos relacionados a testes de desempenho de serviços Web.

A pesquisa foi conduzida, em um primeiro momento, considerando artigos científicos escritos na língua inglesa e foi utilizada a ferramenta ‘google scholar’. Foram feitas duas buscas distintas com as seguintes palavras-chave (em inglês): *performance tests on microservices*; *tests on microservices*. Após essa busca foi encontrado o trabalho de RAHMAN-GAO (2015) e o trabalho proposto por HEORHIADI *et al.* (2016).

Em seguida foi feita uma nova busca com o objetivo de localizar trabalhos com propostas para automatizar testes de desempenho em web services. Nessa busca, foi utilizado como palavra-chave ‘*web service automated testing*’. Essa busca retornou o trabalho de SNEED-VERHOEF, 2013.

5.1.1. Especificação Automática de Testes em Web Services

Os testes em Web Services são, em geral, executados de forma exploratória. A verificação sobre o quanto um teste está adequado para validar um serviço é feita de forma manual e sob responsabilidade da pessoa que o especificou ou que executa o

teste (SNEED-VERHOEF, 2013). Tal limitação deu espaço para o surgimento de maneiras para definir especificações de forma automática e que possam ser interpretadas por processos automatizados.

A especificação automática de testes é ainda mais relevante em uma arquitetura de microservices. Nesse cenário, os serviços passam por ciclos de desenvolvimento de curta duração, sendo que novas mudanças são inseridas com frequência. A grande quantidade de serviços torna o processo de especificação manual dos testes uma tarefa onerosa e suscetível a falhas.

SNEED-VERHOEF (2013) apresentam uma proposta para especificar requisitos em Web Services que operam sob o protocolo SOAP. A proposta permite definir o comportamento esperado através de casos de uso, por meio de uma abordagem centrada no usuário do serviço. Utilizando uma linguagem natural é possível unir a especificação de requisitos com a especificação dos testes, em um único documento. O documento de especificação do serviço SRS (*Service Requirement Specification*) foi definido como um padrão pela ANSI/IEEE (Norma 830). O trabalho apresenta uma extensão da norma, adicionando novas propriedades ao documento padrão: regras, requisitos (funcionais e não funcionais), saídas esperadas, definição de interfaces e os casos de uso.

O trabalho proposto por SNEED-VERHOEF (2013) permite a extração da especificação dos testes a partir do serviço. Além disso, também é possível a execução automática dos testes para validação do serviço. Porém, faz-se necessário utilizar uma linguagem padrão para especificação do serviço. Considerando essa limitação, o presente trabalho contesta que uma abordagem melhor para especificação da requisição de um serviço pode ser a geração automática, durante a implementação do serviço.

É evidente que o trabalho proposto por SNEED-VERHOEF (2013) não tem como propósito principal a especificação dos testes, mas sim a execução de fluxos mais complexos que envolvem requisitos funcionais, não funcionais e até mesmo casos de uso. Para atender o propósito do presente trabalho, apenas a especificação da requisição e posteriormente dos testes é suficiente para a execução dos testes de desempenho.

5.1.2. Ferramenta para Testar a Resiliência de Microservices

O trabalho proposto por HEORHIADI *et al.* (2016) tem como foco o desenvolvimento de um framework para testar a capacidade de resiliência de microservices. O framework parte do pressuposto que os microservices são aplicações fracamente acopladas e que operam através de protocolos simples sobre a rede de comunicação.

O framework, chamado *Gremlin*, permite a um operador a definição de testes e a sua execução através da manipulação das mensagens trocadas pelos microservices na rede de comunicação.

Para que seja possível a execução dos testes de falha, o framework possui agentes que são instalados na infraestrutura onde o microservice é executado. Esses agentes são responsáveis por analisar as requisições que chegam ao serviço e modificar as requisições que são identificadas como requisições de teste. Essa injeção de falhas nas requisições é o que permite a verificação da resiliência do microservice. A resposta para as requisições modificadas é analisada e disponibilizada ao operador em uma interface.

O trabalho possui grande relevância no que diz respeito aos testes de resiliência. O framework desenvolvido é capaz de endereçar diferentes aspectos dos testes de falhas, permitindo ao operador a definição de diversos cenários de teste. Além disso, o framework opera de uma maneira transparente, sem afetar outras requisições do fluxo regular do microservice.

Tendo como pauta a proposta do presente trabalho, que é a execução de testes de desempenho, o framework desenvolvido no trabalho proposto por HEORHIADI *et al.* (2016) não trata esse tipo de teste. O framework é voltado para execução de cenários de falha com o objetivo de verificar a resiliência do microservice, ou seja, trata-se de um propósito diferente do que se busca no presente trabalho.

5.1.3. Arquitetura para Reutilização de Teste de Aceitação em Microservices

O trabalho proposto por RAHMAN-GAO (2015) apresenta uma arquitetura para centralizar testes de aceitação do modelo ágil BDD (*Behavior Driven Development*). O trabalho trata principalmente do reuso dos componentes em uma aplicação composta por microservices.

Através da arquitetura desenvolvida é possível o compartilhamento dos testes de aceitação entre diferentes serviços, pois os testes são armazenados em um repositório central.

Apesar do trabalho tratar de testes de aceitação e não de testes de desempenho, é interessante notar que há um apelo para a melhoria dos testes em uma aplicação formada por microservices.

A abordagem de RAHMAN-GAO (2015) sugere que seja utilizada uma abordagem baseada em *Behavior-Driven Development* (BDD) para construção de testes de aceitação automatizados em microservices.

Nota-se que a abordagem é voltada a testes funcionais, do tipo caixa preta, ou seja, testes de verificação do sistema levando em consideração valores de entrada e valores de saída. Diferentemente do modelo citados, o presente trabalho tem como objetivo testar o requisito não funcional de desempenho. Em comparação com o presente trabalho, percebe-se que buscamos tratar da automatização dos testes de desempenho, ou seja, permitir que esses testes sejam executados por uma aplicação externa sem qualquer intervenção manual. No caso do trabalho proposto por RAHMAN-GAO (2015), o foco é a reutilização dos testes através dos serviços.

5.2. Considerações Finais do Capítulo

O presente capítulo teve como objetivo apresentar os trabalhos correlatos. Nesse contexto, o trabalho de RAHMAN-GAO (2015) que propõe uma arquitetura para reutilização de testes de aceitação em microservices é aquele que mais se assemelha com a proposta deste trabalho. Contudo, a proposta de RAHMAN-GAO tem como foco testes de aceitação enquanto o presente trabalho tem como foco testes de desempenho.

No próximo capítulo será apresentada a arquitetura proposta para automatizar os testes de desempenho em microservices.

6. ARQUITETURA PARA AVALIAÇÃO DO DESEMPENHO DE MICROSERVICES

Este capítulo tem como objetivo apresentar a proposta de arquitetura para avaliar o desempenho de microservices, o principal objetivo deste trabalho. A arquitetura deve ser entendida como uma proposta de organização dos microservices e foi desenvolvida para permitir uma maneira simples e eficaz para a execução dos testes de desempenho em microservices.

6.1. Visão Geral da Arquitetura

A arquitetura proposta divide-se em dois conceitos:

- Uma especificação para permitir que aplicações externas acessem os parâmetros de teste que serão usados durante o teste do serviço. Essa especificação consiste em um conjunto de dados com as informações necessárias para compor as requisições que serão enviadas aos serviços.
- Um mecanismo para anexar e expor a especificação de teste. Esse mecanismo inclui como a especificação deve ser definida no serviço e como aplicações externas podem obter essa especificação.

O aspecto mais significativo na arquitetura proposta é a geração da especificação de teste com todos os parâmetros necessários para a execução do teste de desempenho, e também, o fato dessa especificação estar anexada ao microservice como parte de seu código. Essa composição faz com que cada microservice possua a especificação para testar as suas operações. As especificações de teste permitem a identificação dos limites no que diz respeito a quais operações devem ser testadas para cada serviço. Essa distribuição dos testes permite centralizar os testes de desempenho em uma aplicação de teste, simplificando o gerenciamento e a execução dos testes.

A especificação de teste é obtida através da execução de uma requisição ao serviço utilizando o método HTTP *Options*. A URI (*Uniform Resource Identifier*) a ser utilizada é a mesma designada

para a operação do serviço. A visão geral da arquitetura proposta pode ser encontrada na Figura 12.

O método *Options* é comumente utilizado por um sistema para fornecer informações acerca de seus recursos. O tipo de informação pode incluir: modelos de requisição e resposta, requisitos e a capacidade do servidor. Sendo assim, a solução proposta visa utilizar um método que não é regularmente utilizado pelo serviço para prover suas operações, assim como os métodos *Post* e *Get*. Além disso, o método *Options* pode ser utilizado para fornecer informações acerca da operação do serviço, de modo que a especificação de teste pode ser uma dessas informações.

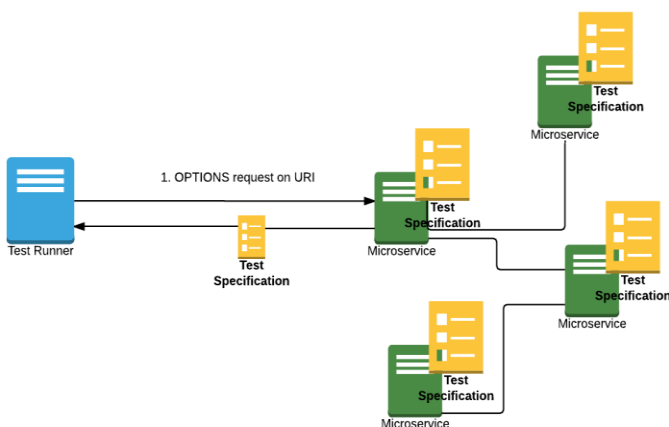


Figura 12 - Visão Geral da Arquitetura Proposta

A Figura 12 apresenta a organização de uma aplicação como um conjunto de microservices. Cada serviço contém a sua especificação de teste será obtida pela aplicação de teste para a execução dos testes de desempenho. A especificação de teste contém as seguintes informações: todos os métodos que estão disponíveis na URI da requisição, um exemplo de parâmetro da requisição, uma descrição que é uma representação semântica da operação e, por fim, dados para validação das respostas recebidas após uma requisição ao serviço.

Juntamente com os dados para a execução do teste de desempenho, o serviço também provê uma representação em *JSON Schema* dos parâmetros da operação requisitada. Essa representação permite uma maior flexibilidade na execução dos

testes, com a possibilidade de modificação dos parâmetros da requisição.

6.2. Especificação dos Testes

Para descrever a forma como os testes são especificados, iremos considerar uma aplicação hipotética que realiza o gerenciamento de transações financeiras. Essa aplicação é composta por um conjunto de microservices que permitem a execução de testes de desempenho de forma automatizada, conforme apresentado pela arquitetura definida na Figura 12. Um dos microservices é responsável por prover operações sobre as transações financeiras. Uma transação financeira é composta pelos seguintes campos: 'ID', 'Country', 'Name', 'Month' and 'Year'. Além disso, também contém uma lista de detalhes, cada qual composto por 3 campos: 'Account', 'AmountUsd' e 'AmountPln'. O campo 'Account' representa uma conta armazenando um valor alfanumérico. O campo 'AmmountUsd' é um valor numérico do tipo Double assim como o campo 'AmmountPln', ambos representa o valor da transação financeira em dólares e dólares e na moeda local, respectivamente.

O microservice de exemplo que realiza operações sobre as transações financeiras possui duas operações: buscar uma transação utilizando um 'ID' e armazenar uma nova transação. Essas operações são utilizadas através de requisições aos métodos *Get* e *Post*, respectivamente na URI '/rest/finance'. Considerando que o microservice possui suporte à execução de testes de desempenho, a especificação para execução dos testes pode ser obtida através de uma requisição que utiliza o método *Options* na URI '/rest/finance'. Tal requisição irá gerar como resultado a especificação apresentada na Figura 13.

```

{
  "POST":{
    "description":"Save a new financial transaction",
    "parameter":{
      "name":"000023989",
      "country":"766",
      "acctMonth":"10",
      "acctYear":"2015",
      "accountLines":      [{
        "amountUsd":1000,
        "amountPlan":1000.0,
        "account":"0083289"}]},
    "parameter-schema":{
      "type":"object",
      "id":"urn:jsonschema:com:finance:company:journal:bean:
        FinancialTransaction",
      "properties":{
        "id":{"type":"string"},
        "name":{"type":"string"},
        "country":{"type":"string"},
        "acctMonth":{"type":"string"},
        "acctYear":{"type":"string"},
        "accountLines":{"type":"array","items":{"type":"object"},
        "id":"urn:jsonschema:com:finance:bean:AccountDetails",
        "properties":{"id":{"type":"string"},
        "amountUsd":{"type":"number"},
        "amountPlan":{"type":"number"},
        "account":{"type":"string"}}}}},
      "validation":{"headerData":{"status":"200"}}
    },
    "GET":{
      "description":"Get transaction by id",
      "parameter":{"parameter":"id","value":"1"},
      "validation":{"bodyData":{"name":"000023989",
        "country":"766","acctMonth":"10","acctYear":"2015"},
        "headerData":{"status":"200"}}
    }
  }
}

```

Figura 13 - Exemplo de Resposta para uma Requisição Options

Na Figura 13 é apresentado a especificação de teste retornada para uma requisição Options em uma URI. A especificação é gerada no formato JSON e contém uma especificação de teste para

cada um dos métodos, no exemplo os métodos POST e GET. Cada especificação de teste contém os seguintes atributos:

- ‘description’: Uma descrição sobre a operação.

- ‘parameter’: O parâmetro a ser usado na requisição de teste, para o método POST esse parâmetro irá compor o corpo da requisição de teste. No caso do método GET esse parâmetro é utilizado na URI.

- ‘parameter-schema’: Representação do parâmetro no formato JSON-Schema, tal representação é útil para que os campos da requisição possam ser redefinidos pelo usuário facilmente na aplicação de teste. O schema pode ser utilizado pela ferramenta de teste para gerar templates de requisição onde o usuário pode inserir os campos já conhecendo os tipos esperados.

- ‘validation’: Dados para validação das resposta recebidas nas requisições de teste. Esses dados permitem que o teste de desempenho seja enriquecido com validações para garantir que o teste foi executado com sucesso. No exemplo fornecido temos a validação para verificar se a resposta retornada possui o status 200.

6.3. Considerações Finais do Capítulo

A arquitetura proposta tem por objetivo permitir a obtenção da especificação para execução dos testes de desempenho de uma maneira simples e eficaz. Com a especificação anexada ao serviço, as aplicações de teste podem obter os parâmetros para execução do teste de uma forma automatizada sem a necessidade de intervenção manual.

A automatização dos testes em uma arquitetura de microservices é de grande importância, pois, com aplicações que podem ser compostas por centenas de serviços a manutenção e execução dos testes de forma manual torna-se inviável ou de alto custo. Além disso, todo processo que requer uma intervenção manual pode estar suscetível a erros, inerentes a qualquer atividade dessa natureza.

A junção entre o serviço e a sua especificação de teste acaba por impor que o serviço esteja sempre em sincronia com a sua especificação de teste. Quando a especificação é mantida de forma

independente – em aplicações de teste como JMeter, por exemplo – o sincronismo deve ser mantido de forma manual. No caso dos microservices, tal tarefa é árdua devido ao grande número de serviços e operações.

7. FPTS – FRAMEWORK FOR PERFORMANCE TEST SPECIFICATION

Com o objetivo de validar a arquitetura proposta foi desenvolvido um framework que implementa alguns aspectos dessa arquitetura para fins de validação da proposta. O framework implementa os seguintes aspectos definidos pela arquitetura:

- Utilização do método HTTP *Options* para retorno da especificação de teste;
- Extração automática da especificação de teste para a operação solicitada; e
- Construção da especificação de teste contendo o modelo de requisição e dados para validação da resposta, conforme especificado na Figura 13.

O principal aspecto no que diz respeito ao funcionamento do framework é a utilização das anotações para geração das especificações de teste. Com as anotações o framework utiliza um filtro capaz de analisar todas as requisições que chegam ao microservice de uma maneira transparente. O framework irá analisar as requisições e, caso seja uma requisição que utiliza o método HTTP *Options*, o framework irá obter do serviço requisitado as informações (verificando as anotações) e, em seguida, irá construir a especificação completa que será retornada como resposta. É possível que a mesma URI seja utilizada para fornecer operações para diferentes métodos HTTP (por exemplo, POST e GET). Nesse caso, o framework irá agrupar as especificações conforme demonstrado na Figura 13, onde a mesma URI é utilizada para os métodos GET e POST.

Para construção da especificação, o framework utiliza como entrada a URI da operação solicitada. Contudo, a mesma URI é também utilizada acesso à operação fornecida pelo serviço (por exemplo, utilizando o método GET). Nesse sentido, o framework é capaz de identificar quando a requisição é enviada para uma das operações do serviço. Nesse caso, o framework simplesmente redireciona a requisição para o serviço atuando de forma transparente ao funcionamento regular do serviço. A Figura 14

apresenta o funcionamento do framework para uma requisição *Options* e para outras requisições.

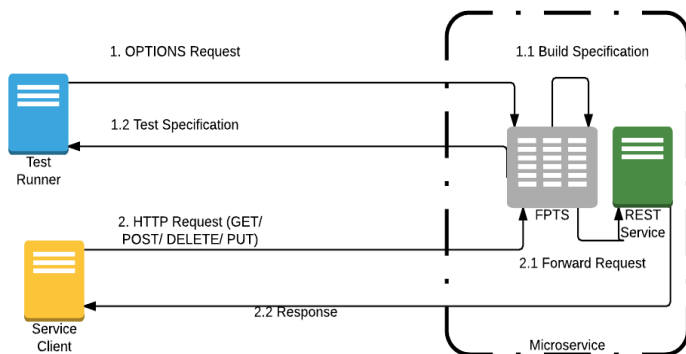


Figura 14 - Comportamento do Framework para Requisições HTTP

A Figura 14 mostra o comportamento do framework para requisições utilizando diferentes métodos do protocolo HTTP. Como evidenciado, o framework irá filtrar apenas requisições que utilizam o método *Options*; para os outros métodos HTTP, o framework irá apenas encaminhar as requisições ao serviço solicitado. Conforme evidenciado pela Figura 14, o framework funciona como uma camada anterior ao serviço do microservice para interceptar as requisições que são enviadas ao serviço.

7.1. Implementação

O framework foi desenvolvido sobre a plataforma *Sprint Boot*⁶ na linguagem Java. O *Spring Boot* fornece diversas funcionalidades e facilidades para a construção de microservices: servidor web embarcado (*Tomcat* ou *Jetty*), framework MVC e frameworks de persistência. De configuração simples, o *Spring Boot* permite a construção de microservices de uma maneira prática e com mínimo esforço. Essas facilidades fazem do *Spring Boot* uma boa escolha de framework para o desenvolvimento de microservices, principalmente para desenvolvedores com experiência na linguagem Java (GREENE, 2015).

⁶ <https://projects.spring.io/spring-boot/>

A implementação do framework tem como foco facilitar a sua utilização, ao mesmo tempo que se mantém transparente ao uso do serviço, permitindo ao desenvolvedor a opção de remover o framework sem grande esforço durante o processo de *build* do microservice. O processo de *build* consiste na compilação e compactação do código em um arquivo instalável no servidor web (por exemplo, um arquivo WAR).

Considerando esses requisitos, a opção foi por utilizar anotações Java que permitem ao desenvolvedor definir os métodos e a forma como as especificações de teste devem ser geradas. As anotações permitem adicionar o framework de uma maneira simples e fácil, ao passo que reduz o acoplamento entre o serviço e o framework. Para dar flexibilidade ao desenvolvedor quanto à inclusão ou não do framework na distribuição do microservice foi utilizada a ferramenta *Maven*⁷ para dividir o framework em dois componentes: uma API e a implementação.

O componente que contém a implementação do framework armazena a implementação do filtro que irá analisar as requisições que chegam ao serviço e compor a especificação de teste, caso se trate de uma requisição utilizando o método HTTP *Options*. Além do filtro, o componente também contém a lógica responsável por construir a especificação com base no conteúdo da anotação utilizada no serviço. Já o componente que contém a API armazena as classes utilizadas na anotação e classes auxiliares utilizadas na definição da especificação, por exemplo, na construção dos parâmetros para validação da resposta.

A separação dos componentes em API e implementação dá flexibilidade ao desenvolvedor para compor diferentes *builds* do serviço, sendo possível optar ou não pela inclusão do framework na distribuição do microservice sem que sejam necessárias modificações no código. Durante a execução do microservice, a API do framework irá tentar iniciar a classe provedora do filtro que está presente no componente que contém a implementação. Caso a classe do filtro não seja encontrada, as

⁷ <https://maven.apache.org/>

funcionalidades do framework são desabilitadas sem que seja necessária qualquer configuração por parte do desenvolvedor do microservice. Para que seja possível manter a separação entre a API e a implementação é utilizado o recurso da reflexão, onde a API faz a tentativa de iniciar a classe de filtro sem que seja mantida uma referência direta para essa classe. Através desse recurso é possível a remoção do framework sem que seja necessária a alteração do código do microservice.

A técnica de reflexão é também utilizada pelo componente de implementação do framework para extração da especificação de teste do serviço, o que torna desnecessária a existência de uma conexão entre o componente de implementação e o serviço.

7.2. Forma de Utilização

Sob a perspectiva do desenvolvedor, o framework permite a exposição das especificações de teste com um mínimo de esforço, através de anotações Java nos métodos que retornam as especificações. Para que seja possível utilizar as anotações que definem os métodos de retorno das especificações de teste, o desenvolvedor deve adicionar ao seu projeto o componente da API utilizando o jar ou através do controle de dependências do *Maven*. Com a API disponível, o desenvolvedor irá implementar os métodos para retornar à especificação de teste, devendo haver um método para cada operação do serviço. O último passo é adicionar o filtro do framework aos filtros do *Spring Boot*, o que fará com que o framework possa interceptar todas as requisições que chegam ao serviço.

Considerando o mesmo exemplo que é apresentado na seção 6 (aplicação para gerenciamento de transações financeiras) e cujo modelo de especificação de teste é apresentado na Figura 13, digamos que estamos implementando a especificação de teste para a operação *save* (método POST). No exemplo citado, esse método é solicitado através da URI `'rest/finance'`. Para expor a especificação de teste para essa operação devemos criar um novo método, anotando o mesmo com `'@PerformanceTest'` conforme apresentado na Figura 15.

```

@PerformanceTest(path = "/rest/finance",
    httpMethod = HttpMethodEnum.POST,
    description = "Save a new financial
    transaction")
public TestSpec<FinancialTransaction>
getTestSpecForSave() throws IOException
{
    FinancialTransaction testObj =
        new
FinancialTransaction("000023989",
                    "766", "10", "2015");
    AccountDetails accDetails =
        AccountDetails.build("0083289",
1000d,
                    1000d);
    testObj.setAccLines(accDetails);
    TestValidationsBuilder
validationBuilder =
        new TestValidationsBuilder();
    TestSpec<FinancialTransaction>
testSpec =
        new TestSpec<FinancialTransaction>
(testObj, validationBuilder.

```

Figura 15 - Uso do Framework para Definir uma Especificação de Teste

A anotação é a forma utilizada pelo framework para inferir quais operações do serviço devem fornecer as especificações de teste. O método deve sempre ter como retorno um objeto da classe 'TestSpec', o conteúdo dessa classe é apresentado na Figura 16.

```

public class TestSpec<T extends Serializable> {
    private Serializable testParameter;
    private TestValidationsBuilder validationData;

    public TestSpec(Serializable testParameter, TestValidationsBuilder validationData) {
        super();
        this.testParameter = testParameter;
        this.validationData = validationData;
    }

    public Serializable getTestParameter() {
        return this.testParameter;
    }

    public TestValidationsBuilder getValidationData() {
        return this.validationData;
    }
}

```

Figura 16 - Classe Utilizada para Geração da Especificação de Teste

Esse objeto irá conter os dados necessários para compor a especificação de teste completa, conforme apresentado na Figura 13. Como o exemplo apresentado é referente a uma operação que utiliza o método POST, o objeto que contém a especificação de teste inclui também o parâmetro a ser utilizado na requisição. No exemplo apresentado, esse parâmetro é um objeto da classe 'FinancialTransaction'.

Juntamente com o modelo de requisição, o objeto que contém a especificação também armazena os dados para validação das respostas ao serviço. No exemplo apresentado, há uma validação para checar se o cabeçalho retornado na resposta tem status 200 (OK).

A anotação '@PerformanceTest' possui três atributos:

- path: define o caminho onde o serviço provê a operação; esse valor será utilizado pela aplicação de testes para enviar as requisições.
- httpMethod: define o método HTTP que deve ser utilizado para solicitar a operação ao serviço; seu valor é definido como uma enumeração, podendo ser qualquer um dos valores aceitos pela RFC 7231⁸.
- description: um valor semântico para a operação realizada pelo serviço.

As informações obtidas na anotação, juntamente com o objeto retornado pelo método, serão utilizados pelo framework para composição da especificação de teste. Caso haja mais de uma

⁸ <https://tools.ietf.org/html/rfc7231#section-4.3>

especificação com o mesmo valor de ‘path’, o framework irá construir a especificação como uma composição dos métodos. Um exemplo pode ser observado na Figura 13.

No exemplo apresentado na Figura 15, os valores dos parâmetros do objeto de teste são colocados diretamente no código apenas para título de demonstração dos valores, para que seja possível fazer uma conexão entre a definição da especificação e o resultado gerado (Figura 13). O recomendado é que esses valores sejam retirados dinamicamente de uma base dados quando uma requisição ao serviço for feita.

7.3. Principais Funcionalidades

O propósito do framework é prover uma maneira simples e eficaz para que o desenvolvedor possa integrar o mecanismo de especificação dos testes de desempenho ao microservice. O resultado final foi bastante promissor, devendo simplificar de forma significativa o processo de realização de testes em microservices, tendo em vista que o framework possui diversas funcionalidades que auxiliam na definição da especificação de testes:

- 1) API: Permite ao desenvolvedor a remoção do framework durante o processo de *build* da aplicação. Essa funcionalidade é particularmente útil quando a aplicação é instalada em um ambiente de produção e a especificação de teste não deve ser exposta.
- 2) Anotações: Permitem a definição das especificações de teste de uma maneira simples e sem demandar grandes alterações no código do serviço. Também dão flexibilidade ao desenvolvedor em expor apenas as operações do serviço que podem ser alcançadas pela aplicação de teste. As anotações são de grande eficácia para reduzir o acoplamento entre o serviço e o framework.
- 3) Dados Dinâmicos para Teste: Refere-se à possibilidade de utilizar uma base de dados para obter os dados que serão usados na composição da requisição da especificação de teste. Utilizando esses dados dinâmicos, o desenvolvedor pode utilizar dados

atualizados na execução dos testes de desempenho. Um exemplo pode ser a geração de uma especificação para uma operação que utiliza o método *Get* para retornar um objeto pelo seu ID na base de dados. O desenvolvedor pode obter esse objeto diretamente da base de dados e, no momento de construção da especificação, esse objeto terá seu valor atualizado com o valor presente na base de dados. Dessa forma, os dados utilizados no teste estão sempre em sincronia com o modelo presente na base de dados sem que seja necessário atualizar a especificação de teste caso ocorram alterações no modelo de dados.

- 4) **Dados para Validação:** Durante os testes de desempenho é importante também executar validações nos dados retornados pelo serviço a fim de garantir que o teste foi executado com sucesso. Dessa forma, o framework permite a construção desses elementos de validação das respostas de uma maneira simples, por exemplo, permite a validação do cabeçalho de resposta verificando se o *Status* é 200 (OK). No exemplo apresentado na Figura 15, além do *Status*, é criada uma validação para o corpo da resposta HTTP.
- 5) **Analisador de Anotações:** Essa funcionalidade permite a validação das anotações em tempo de compilação. Dessa forma, durante o processo de *build* haverá a validação do método com a anotação ‘PerformanceTest’ para verificar se o seu retorno é do tipo ‘TestSpec’ e se a classe de retorno do método possui o tipo genérico (que representa o tipo que deve ser utilizado na requisição ao serviço). Caso a validação identifique algum erro, uma mensagem será apresentada e o processo de *build* será interrompido. Na Figura 17 há um exemplo no qual a classe utilizada no retorno do método não possui o tipo genérico, resultando em um erro de compilação.

```
Caused by:  
org.apache.maven.plugin.compiler.CompilationFailureException:  
Compilation failure  
Returning type must have the generic class. Please set:  
TestSpec<Class> as the returning type at  
org.apache.maven.plugin.compiler.AbstractCompilerMojo.execute\(Ab  
stractCompilerMojo.java:858\)  
    at  
    org.apache.maven.plugin.compiler.CompilerMojo.execute(CompilerMo  
jo.java:129)  
    at  
    org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo(De  
faultBuildPluginManager.java:134)  
    at  
    org.apache.maven.lifecycle.internal.MojoExecutor.execute(MojoExe  
cutor.java:208)  
    ... 20 more
```

Figura 17- Exemplo de Erro de Compilação Durante a Validação das Anotações.

8. AVALIAÇÃO

Para avaliar o framework utilizamos um microservice de uma aplicação de exemplo. A aplicação utilizada como exemplo é uma aplicação financeira e foi escolhida uma operação do microservice. A aplicação de exemplo possui uma operação para adicionar novas transações financeiras e outra operação para recuperar as operações já adicionadas. A operação utilizada na avaliação é a operação para salvar uma nova transação financeira, apresentada na Figura 15. A especificação de teste gerada para essa operação é apresentada na Figura 13.

A avaliação foi dividida em duas etapas. A primeira etapa da avaliação tem como objetivo identificar se o framework pode causar algum impacto no desempenho do serviço. É importante realizar essa avaliação para certificar que o desempenho obtido por uma ferramenta de teste é de fato o desempenho do serviço e que o framework não causa prejuízo caso esteja operando juntamente com o serviço. A segunda etapa da avaliação tem como objetivo verificar o tempo gasto pelo framework para gerar uma especificação de teste.

Como parte do funcionamento do framework, todas as requisições ao serviço são capturadas e analisadas. Caso seja uma requisição com o método Options, o framework irá fornecer a especificação de teste. Já se for uma requisição com qualquer outro método, o framework irá encaminhar a requisição ao serviço. Dessa forma, como todas as requisições são interceptadas pelo framework, é importante verificar se há algum impacto no desempenho do serviço quando o framework está sendo utilizado.

8.1. Primeira Etapa – Avaliação do Impacto do Framework

O objetivo dessa etapa da avaliação é identificar se o framework pode causar algum impacto no desempenho do serviço. Consideramos que o serviço irá prover suas operações através dos métodos do padrão HTTP, no exemplo utilizamos o método POST.

8.1.1. Metodologia

Para a avaliação, consideramos o número de amostras como o número de requisições enviadas ao serviço. Para obter o número de amostras a serem utilizadas foi desenvolvido um teste piloto para obter o desvio padrão médio. Para o teste piloto utilizamos o microservice com o framework, o que foi decidido através de uma escolha aleatória dentre as opções com ou sem framework.

A configuração de carga utilizada para o teste piloto foi de 20 *threads* com 50 requisições cada, resultando em um total de 1.000 requisições ao serviço. O resultado do teste piloto é apresentado na Figura 18.

Para o tempo de resposta, o resultado foi uma média de 226ms e o desvio padrão foi de 275. Além disso, utilizamos um nível de confiança de 99%, o que gerou um valor para 'Z' de 1,96 e a margem de erro de 0,5%. Com base nesses valores, utilizamos a seguinte fórmula para obter o tamanho necessário para a amostra:

$$\text{Tamanho da Amostra} = \frac{Z * \text{desvioPadrão}}{\text{margemDeErro}}$$

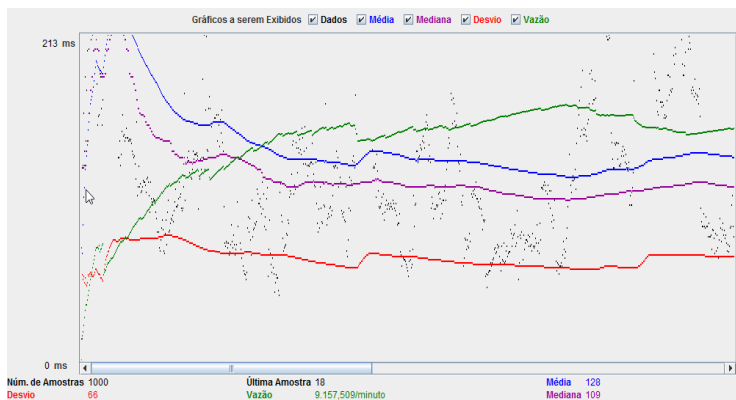


Figura 18 - Resultados do Teste Piloto

Após a aplicação da fórmula, foi constatado que o tamanho necessário da amostra seria de aproximadamente 10.000. Para melhor simular uma situação real, onde as requisições partem de diversos clientes, o número de amostras foi dividido em 25 clientes, sendo que cada cliente enviou um número total de 400 requisições, resultando no total de 10.000 requisições.

Para evitar variações indesejadas – por exemplo, problemas na rede de comunicação – cada teste foi executado duas vezes, e o valor final do teste é obtido através da média das duas rodadas de teste. Para definir a ordem dos testes foi feito um sorteio com os dois valores possíveis: WF (com framework) e NF (sem o framework). O resultado final foi: WF, NF, NF e WF.

8.1.2. Configuração e Execução

Com o objetivo de evitar possíveis variações causadas por diferenças na infraestrutura utilizada para executar as aplicações, foram utilizadas instâncias da aplicação na plataforma *Amazon EC2*. A instância utilizada foi do tipo t1 micro (1 vCPU, 3.75GB de memória e disco SSD para armazenamento).

Cada teste foi executado com apenas uma instância da aplicação sendo executada por vez. Além disso, a máquina virtual utilizada possuía apenas os processos do sistema operacional e nenhuma outra aplicação foi executada em paralelo.

A avaliação baseia-se em dois parâmetros: vazão (requisições/segundo) e tempo de resposta (em milissegundos). Os testes foram executados utilizando a ferramenta JMeter.

O procedimento de teste consistiu em enviar uma requisição utilizando o método Post, tendo como conteúdo um objeto ‘FinancialTransaction’. Esse objeto é o mesmo apresentado na Figura 15 e identificado como a variável ‘testObj’. No microservice, o objeto recebido na requisição é inserido em uma base de dados em memória (para evitar gargalos do banco de dados) e o mesmo objeto com um parâmetro ‘ID’ que foi gerado pela base de dados é retornado ao cliente.

8.1.3. Resultados

Os resultados obtidos nos testes são apresentados na Tabela 1 na ordem em que foram executados.

A Tabela 2 contém a média para a vazão e para o tempo de resposta em cada uma das rodadas de teste (média para as rodadas do teste WF e a média para as rodadas do teste NF).

Tabela 1 - Resultado dos Testes

Teste	Resultados para cada Rodada de Teste		
	<i>Média Tempo Resposta (ms)</i>	<i>Vazão Média (req. por segundo)</i>	<i>Desvio Padrão</i>
1 – WF	33	727	16
2 – NF	32	757	17
3 - NF	39	614	21
4 - WF	28	846	12

Tabela 2 - Média para os Testes WF e NF

Teste	Média para os Testes WF e NF	
	<i>Média Tempo Resposta (ms)</i>	<i>Vazão Média (req. por segundo)</i>
1 – WF	30,5	785,5
2 - NF	35,5	685,5

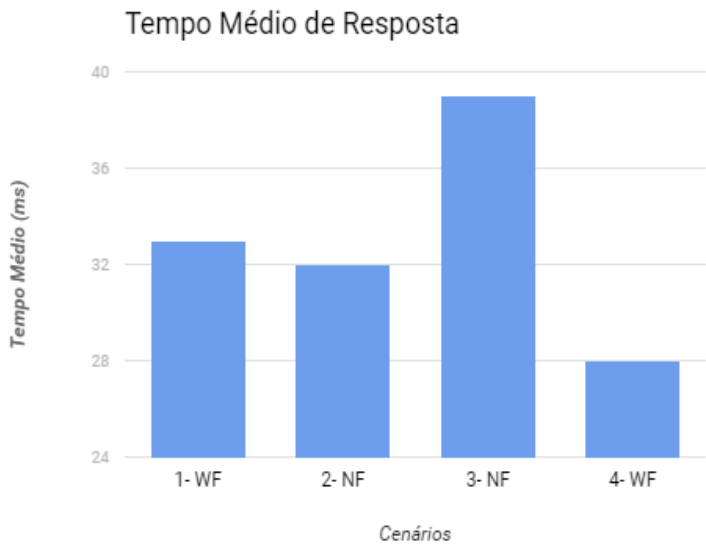


Gráfico 1 – Resultados para o Tempo Médio de Resposta

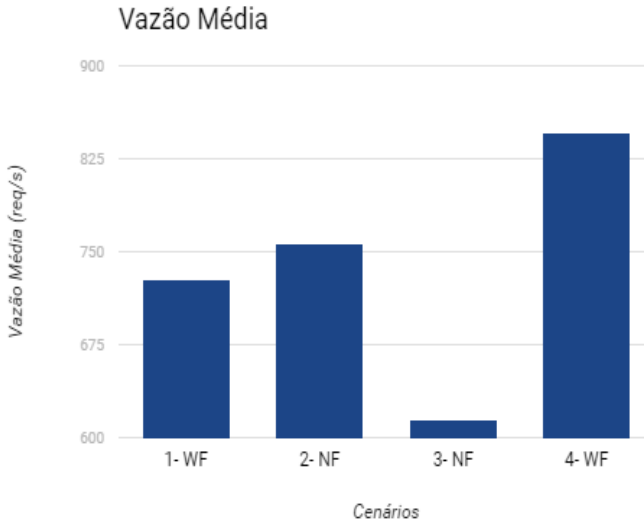


Gráfico 2 – Resultados para a Vazão Média

Os gráficos 1 e 2 mostram os resultados das tabelas para o tempo médio de resposta e a vazão média na forma de um gráfico de barras, respectivamente.

Com base nos resultados podemos afirmar que o framework não causa qualquer impacto no desempenho do microservice.

Conforme apresentado na Tabela 2, a média do tempo de resposta para o microservice utilizando o framework é menor que a média sem o framework.

Considerando a forma como o framework foi desenvolvido, os resultados obtidos eram esperados, uma vez que para todas as requisições que não utilizam o método *Options* o framework irá apenas encaminhá-las ao serviço. Nenhum processamento adicional é feito por parte do framework além da avaliação do método utilizado pela requisição.

A ausência de impacto na utilização do framework permite que o mesmo seja utilizado em qualquer ambiente, inclusive em um ambiente de produção. O framework poderia ser utilizado como um mecanismo para rastreamento de gargalos de desempenho no ambiente de produção. Adicionalmente, o provedor do serviço pode utilizar o framework como uma ferramenta para que os seus clientes possam testar o desempenho do framework e assim aferir se os seus parâmetros de QoS que foram contratados estão sendo

atendidos. Esse mecanismo pode ser utilizado para dar os clientes uma maior garantia para os serviços contratados, uma vez que o próprio cliente pode testar o serviço de uma maneira simples e eficaz utilizando a especificação de teste que é fornecida pelo próprio serviço.

Em geral, os resultados da avaliação foram bastante promissores no sentido de comprovar que o framework não causa impacto nas operações fornecidas pelo microservice e que pode ser utilizado sem limitações.

8.2. Segunda Etapa – Avaliação da Geração da Especificação de Teste

Nessa etapa avaliamos o framework quanto ao tempo gasto para gerar as especificações de teste. Foi utilizada a mesma operação apresentada na Figura 15. A especificação de teste gerada para essa operação é apresentada na Figura 13.

Para este teste foram enviadas três requisições ao serviço, utilizando o método OPTIONS e foi contabilizado o tempo total gasto do momento que a requisição é enviada da aplicação cliente até o recebimento da especificação de teste na aplicação cliente.

Para garantir que o teste foi efetuado com sucesso foi verificado o cabeçalho de resposta e também o conteúdo da resposta para verificar se a especificação foi de fato enviada pelo serviço.

8.2.1. Execução e Resultados

Para a execução dos testes foi utilizada a ferramenta JMeter. A aplicação foi executada na mesma infraestrutura utilizada na etapa 1.

Primeiramente, foi executado um teste piloto nos mesmos moldes daquele executado na etapa 1, com o objetivo de verificar o tamanho necessário da amostra. O teste piloto consistiu em 3 testes executados separadamente.

A Figura 19 mostra o resultado apresentado pelo JMeter para a execução do primeiro teste. Na Figura 20 é apresentada a resposta retornada pelo serviço. Dentre os parâmetros apresentados destaca-

se a latência, ‘*Body size in bytes*’ que é o tamanho dos dados da resposta e também o código da resposta que indica se o retorno é um código de sucesso (200).

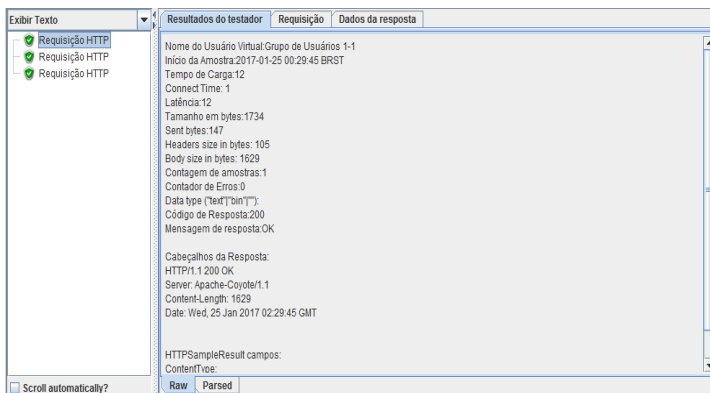


Figura 19 - Resultado para o Primeiro Teste

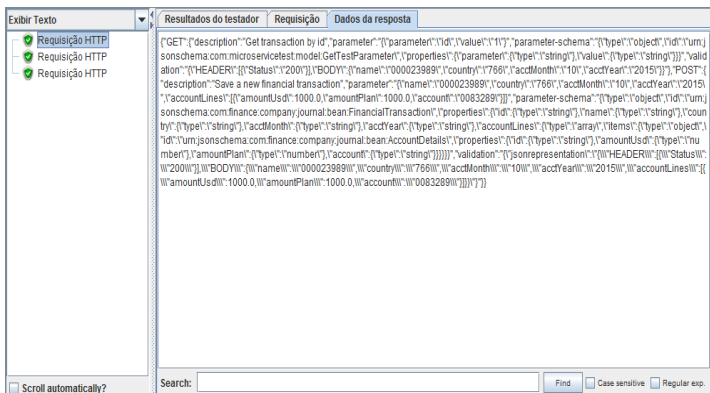


Figura 20 - Resposta do Serviço para o Primeiro Teste

A seguir é apresentado o tempo gasto para cada uma das requisições em milissegundos.

Tabela 3 - Tempo de Resposta para Cada Teste

Teste	Tempo de Resposta (ms)
1	12
2	15
3	8

A média para o tempo de resposta foi de 11.6 milissegundos e o desvio padrão foi de 3.5.

Com base nos resultados obtidos no teste piloto utilizamos a fórmula para calcular o tamanho necessário da amostra com o valor de z igual a 1,96 e margem de erro de 0,5%.

$$\text{Tamanho da Amostra} = \frac{Z * \text{desvioPadrão}}{\text{margemDeErro}}$$

O resultado final para o tamanho da amostra foi de 13,72. Para a avaliação consideramos o tamanho da amostra como 14.

Foram executadas duas rodadas de teste enviando 14 requisições para obtenção da especificação de teste em cada uma das rodadas. A Figura 21 mostra os resultados para a primeira rodada. A Tabela 4 apresenta a média do tempo de resposta e o desvio padrão para as duas rodadas.



Figura 21 - Resultado para a Primeira Rodada

Tabela 4 - Resultado da Avaliação da Geração da Especificação de Teste

Rodada	Média Tempo de Resposta (ms)	Desvio Padrão
1	11	4
2	12	8

Os resultados mostraram que o framework é capaz e gerar a especificação de teste em um tempo relativamente baixo quando comparado ao tempo gasto por exemplo para a execução de uma operação POST do serviço. A média do tempo de resposta do serviço com o framework foi de 30.5ms (resultado apresentado na etapa 1) já o tempo médio para geração da especificação de teste foi de 11.5ms.

É evidente que a operação POST possui um parâmetro que é enviado juntamente com a requisição ao passo que no método OPTIONS não há requisição. Porém, o tamanho dos dados retornados pelo serviço na resposta é consideravelmente maior para a requisição OPTIONS (1629 bytes) quando comparada a requisição utilizando o método POST (132 bytes). Tal fator (o tamanho dos dados) impacta diretamente no tempo de resposta.

9. CONCLUSÕES E TRABALHOS FUTUROS

Microservices emergiram como uma alternativa ao desenvolvimento de aplicações para superar as dificuldades encontradas no modelo de desenvolvimento tradicional, conhecido como aplicações monolíticas. Os microservices são naturalmente desacoplados, permitindo uma maior flexibilidade na instalação, desenvolvimento, escalabilidade e na adoção de tecnologias heterogêneas. Por outro lado, essa flexibilidade acaba por gerar alguns desafios em algumas etapas do desenvolvimento da aplicação, como a fase de testes, que é o foco deste trabalho. Considerando testes de desempenho, as aplicações que utilizam a arquitetura de microservices podem ser compostas por um grande conjunto de serviços (na ordem de centenas), o que torna o gerenciamento e controle manual dos testes uma tarefa árdua ou até impraticável.

Com o objetivo de simplificar os testes em uma aplicação composta por um conjunto de microservices, foi desenvolvido no presente trabalho um modelo arquitetural para permitir a automatização dos testes de desempenho nessas aplicações.

A arquitetura foi desenvolvida para permitir uma integração simples com os microservices, requerendo apenas pequenas alterações na aplicação. A principal funcionalidade da proposta desenvolvida é a adição das especificações de teste ao próprio microservice. Dessa forma, a consistência entre o microservice e seus testes é sempre mantida. Mudanças nas operações do serviço demandam mudanças nos testes, e mantém a especificação sempre em sincronia com o microservice. Foi utilizado o método HTTP Options para expor a especificação de teste e permitir a integração de aplicações externas.

Além da arquitetura, foi desenvolvido um framework que implementa algumas das principais funcionalidades presentes na arquitetura. O framework foi testado em uma avaliação de desempenho, utilizando uma aplicação da área financeira como exemplo. A avaliação permitiu aferir que o framework não tem impacto direto no desempenho do microservice.

De um modo geral, o estudo desenvolvido no presente trabalho é bastante promissor. A arquitetura proposta demonstrou ser simples e eficiente para a automatização dos testes de desempenho em uma aplicação composta por microservices. O framework desenvolvido é de fácil integração, mantendo um baixo acoplamento e utilizando anotações Java para prover suas funcionalidades. A avaliação realizada sobre framework mostra que o mesmo pode ser utilizado em qualquer ambiente sem causar prejuízo ao desempenho do microservice. Além disso, o framework é capaz de gerar a especificação de teste em um tempo extremamente baixo quando comparado ao tempo gasto pelo serviço para realizar as suas operações.

Como um próximo passo, é interessante a evolução do framework desenvolvido. Pode ser integrada uma API à implementação, com o intuito de facilitar a integração das aplicações cliente com ferramentas para execução dos testes de desempenho, como o JMeter. Também pode ser integrado um repositório central para armazenar os testes de desempenho mantendo-os centralizados em um único local. Esse repositório pode também ser utilizado para armazenar dados históricos acerca de todos os testes de desempenho realizados para cada operação de cada microservice, permitindo a identificação de gargalos ou o impacto de novas mudanças adicionadas a cada um dos microservices.

Um bom uso para o framework é a adição de uma biblioteca de execução de testes do serviço. Essa melhoria permitirá a execução dos testes de desempenho diretamente no ambiente onde o serviço é executado, eliminando, assim, possíveis variações geradas por uma infraestrutura ineficiente no cliente de testes ou na rede de comunicação. Esse tipo de teste permite a extração de métricas de desempenho mais realistas e que representam a real capacidade do serviço. Para a construção dessa biblioteca, pode ser utilizada a tecnologia de contêiner, como o Docker, para prover um mecanismo leve e independente para o gerenciamento dos clientes que executam as requisições ao microservice.

As funcionalidades para testes funcionais podem também ser expandidas. Com essa melhoria, um grande conjunto de validações podem ser adicionadas à especificação de teste no momento da sua construção. Os testes funcionais permitiram a validação do microservice sob a ótica dos seus requisitos funcionais e o

framework pode ser utilizado como uma solução completa para testes automatizados em microservices.

Por fim, é importante portar o framework para outras linguagens de programação que são comumente utilizadas na construção de microservices, como Go e Python. A disponibilização do framework em outras linguagens de programação é de grande importância devido à premissa de que uma arquitetura de microservices é heterogênea em termos de linguagens de programação. Dessa forma, é reforçada uma das grandes vantagens da arquitetura de microservices, que é a possibilidade de utilizar a linguagem de programação mais adequada para cada funcionalidade da aplicação.

REFERÊNCIAS

- ALONSO, Gustavo et al. **Web services**. Springer Berlin Heidelberg, 2004.
- APPLE, Lauria. **Making Architecture Work in Microservice**. Disponível em: <<http://tech.gilt.com/post/102628539834/making-architecture-work-in-microservice>>. Acesso em: 24/05/2014.
- BARTOL, Natalia, COADY, Gary, TRENAMAN, Adrian. **Deploying Microservices to AWS at Gilt: Introducing ION-Roller**. Disponível em: <http://www.infoq.com/articles/gilt-deploying-Microservices-aws?utm_source=infoq&utm_medium=popular_widget&utm_content=article&utm_campaign=popular_content_list>. Acesso em: 16/05/2015.
- BEAL, Vangie. **Client-server architecture**. Disponível em: <http://www.webopedia.com/TERM/C/client_server_architecture.html>. Acesso em: 16/05/2015.
- BERNSTEIN, Philip A.; NEWCOMER, Eric. **Principles of transaction processing**. Morgan Kaufmann, 2009.
- BONÉR, Jonas; FARLEY, Dave; KUHN, Roland; THOMPSON, Martin. **The Reactive Manifesto**. Disponível em: <<http://www.reactivemanifesto.org>>. Acesso em: 29/05/2015.
- BRYANT, Daniel. **Scaling Microservices at Gilt with Scala, Docker and AWS**. Disponível em: <<http://www.infoq.com/news/2015/04/scaling-Microservices-gilt>>. Acesso em: 19/05/2015.
- CHAFEE, Alex. **What is a web application (or "webapp")?**. Disponível em: <<http://www.jguru.com/faq/view.jsp?EID=129328> >. Acesso em: 29/05/2015.
- CONALLEN, Jim. **Building Web applications with UML**. Addison-Wesley Longman Publishing Co., Inc., 2002.
- DVORKIN, Eugene. **Seven micro-services architecture problems and solutions**. Disponível em:

<<http://eugenedvorkin.com/seven-micro-services-architecture-problems-and-solutions>>. Acesso em: 01/06/2015.

DOCKER. Docker, Inc. <<https://www.docker.com>>. Acesso em: 01/07/2015.

FIELDING, R. T. **REST: Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doctoral dissertation) - University of California, Irvine, 2000. Disponível em: <<http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf>>. Acesso em: 05/06/2015.

FOWLER, Martin. **Patterns of enterprise application architecture**. Addison-Wesley Longman Publishing Co., Inc., 2003. 533 p. ISBN 0321127420

FOWLER, Martin, LEWIS, James. **Microservices**. Disponível em: <<http://martinfowler.com/articles/microservices.html>>. Acesso em: 05/05/2015.

GABRIELSON, Heidi. **Top 10 reasons to use synthetic monitoring**. Disponível em: <<http://www.riverbed.com/blogs/Top-10-reasons-to-use-synthetic-monitoring.html>>. Acesso em: 10/05/2015.

RAHMAN, Mazedur; GAO, Jerry. **A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development**. In: Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on. IEEE, 2015. p. 321-325.

GISOLFI, Dan. **Web services architect, Part 2: Models for dynamic e-business**. Disponível em: <<http://www.ibm.com/developerworks/library/ws-arc2/>>. Acesso em: 02/06/2015.

GREENE, Dan. **Building a Microservice Architecture with Spring Boot and Docker, part I**. Disponível em: <<http://www.3pillarglobal.com/insights/building-a-microservice-architecture-with-spring-boot-and-docker-part-i>>. Acesso em: 12/09/2016.

HEORHIADI, Victor et al. Gremlin: Systematic Resilience Testing of Microservices. In: **Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on**. IEEE, 2016. p. 57-66.

HOFF, Todd. **Microservices - Not A Free Lunch!**. Disponível em: < <http://highscalability.com/blog/2014/4/8/Microservices-not-a-free-lunch.html>>. Acesso em: 06/05/2015.

HOORN, André Van et al. **Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis**. *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012. p. 247-248.

HURWITZ, Judith et al. **Service Oriented Architecture For Dummies**. Indianapolis: Wiley Publishing, Inc., 2007.

IBM. **Web Services architecture overview**. Disponível em: <<http://www.ibm.com/developerworks/webservices/library/w-ovr/>>. Acesso em 01/06/2015.

ITU-T. **E.800: Terms and definitions related to quality of service and network performance including dependability**. *ITU-T Recommendation*. Agosto 1994. Acesso em 13/06/2015.

JANSSEN, Cory. **Application Monitoring**. Disponível em: < <http://www.techopedia.com/definition/29133/application-monitoring>>. Acesso em: 08/06/2015.

KARATSUBA, A. OFMAN, Yu. **Multiplication of Many-Digital Numbers by Automatic Computers**. *Proceedings of the USSR Academy of Sciences* 145: 293–294, 1963.

KRAFZIG, Dirk; BANKE, Karl; SLAMA, Dirk. **Enterprise SOA: service-oriented architecture best practices**. Prentice Hall Professional, 2005.

LUDWIG, Heiko et al. A service level agreement language for dynamic electronic services. **Electronic Commerce Research**, v. 3, n. 1-2, p. 43-59, 2003.

MANI, Anbazhagan; NAGARAJAN, Arun. **Understanding quality of service for Web services**. Disponível em: <<https://www.ibm.com/developerworks/library/ws-quality/>>. Acesso em: 11/06/2015.

MYERSON, Judith. **Use SLAs in a Web services context, Part 1: Guarantee your Web service with a SLA**. Disponível em: <<http://www.ibm.com/developerworks/library/ws-sla/>>. Acesso em 12/06/2015.

NAMIOT, Dmitry; SNEPS-SNEPPE, Manfred. On Microservices Architecture. **International Journal of Open Information Technologies**, v. 2, n. 9, p. 24-27, 2014.

NATIONS, Daniel. **Web Applications, What is a Web Application?**. Disponível em: <http://webtrends.about.com/od/webapplications/a/web_applications.htm>. Acesso em: 03/05/2015.

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

ORACLE, 2013. **The JAVA EE 6 Tutorial**. Disponível em <<https://docs.oracle.com/javaee/6/tutorial/doc/gjvvh.html>>. Acesso em: 05/06/2015.

RAJENDRAN, T.; BALASUBRAMANIE, P. **Analysis on the study of QoS-aware web services discovery**. arXiv preprint arXiv:0912.3965, 2009. Disponível em: <<http://arxiv.org/abs/0912.3965>>. Acesso em: 12/06/2015.

RAJIV, Mordani. **JAVA Servlet Specification v3.0**. Disponível em: < http://download.oracle.com/otn-pub/jcp/servlet-3.0-fr-eval-oth-JSpec/servlet-3_0-final-spec.pdf>. Acesso em: 27/05/2015.

RICHARDSON, Chris. **Pattern: API Gateway**. Disponível em: < <http://microservices.io/patterns/apigateway.html>>. Acesso em: 26/06/2015.

SATO, Danilo. **CanaryRelease**. Disponível em: < <http://martinfowler.com/bliki/CanaryRelease.html>>. Acesso em: 08/06/2015.

SHACKLETT, Mary E. **Five Key Points for Every SLA**. Disponível em: <<http://archive.is/mtN2O#selection-1429.0-1429.29>>. Acesso em: 14/06/2015

SIMONE, Sergio De. **Practical Implications of Microservices in 14 Tips**. Disponível em: <http://www.infoq.com/articles/Microservices-practical-tips?utm_source=infoq&utm_medium=related_content_link&utm_campaign=relatedContent_articles_clk>. Acesso em: 10/05/2015.

SONI, Ketan; MISTRY, Jesal. **Testing Strategies for Microservices Architecture**. Disponível em <<http://pt.slideshare.net/ThoughtWorks/testing-strategies-for-micro-services>>. Acesso em: 28/06/2015.

STENBERG, Jan. **Experiences from Failing with Microservices**. Disponível em:

<<http://www.infoq.com/news/2014/08/failing-microservices>>.

Acesso em: 17/05/2015.

SNEED, Harry M.; VERHOEF, Chris. **Natural language requirement specification for web service testing**. In: Web Systems Evolution (WSE), 2013 15th IEEE International Symposium on. IEEE, 2013. p. 5-14.

THAKARE, Sheetal; CHAVAN, Savita; CHAWAN, P. M. **Software Testing Strategies and Techniques**. International Journal of Emerging Technology and Advanced Engineering, 2012.

TOFFETTI, Giovanni et al. **An architecture for self-managing microservices**. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud. ACM, 2015. p. 19-24.

YANG, Stephen JH et al. **Service-level agreement-based QoS analysis for web services discovery and composition**. International Journal of Internet and Enterprise Management, v. 5, n. 1, p. 39-58, 2006.