UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

THIAGO BARONIO TREVISAN

APLICABILIDADE DO FRAMEWORK REST ASSURED SOBRE UM WEB SERVICE DE UMA EMPRESA REAL

TRABALHO DE CONCLUSÃO DE CURSO

MEDIANEIRA

2016

THIAGO BARONIO TREVISAN

APLICABILIDADE DO FRAMEWORK REST ASSURED SOBRE UM WEB SERVICE DE UMA EMPRESA REAL

Trabalho de Conclusão de Curso apresentado ao Departamento Acadêmico de Computação da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Bacharel em Computação".

Orientador: Prof. Marcio Angelo Matté

MEDIANEIRA

2016

TERMO DE APROVAÇÃO

Aplicabilidade do Framework Rest-Assured sobre um Web Service de uma Empresa Real

Por

Thiago Barônio Trevisan

Este Trabalho de Conclusão de Curso (TCC) foi apresentado às 14:00h do dia 17 de Junho de 2016 como requisito parcial para a obtenção do título de no Curso Bacharelado em Ciência da Computação, da Universidade Tecnológica Federal do Paraná, *Câmpus* Medianeira. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Marcio Angelo Matté
UTFPR – Câmpus Medianeira
(Orientador)

Prof. Alessandra Bortoletto Garbelotti Hoffmann
UTFPR – Câmpus Medianeira
(Convidado)

Prof. Jorge Aikes Júnior
UTFPR – Câmpus Medianeira
(Convidado)

Prof. Jorge Aikes Junior
UTFPR – Câmpus Medianeira
(Responsável pelas atividades de TCC)

RESUMO

TREVISAN, Thiago B. APLICABILIDADE DO FRAMEWORK REST ASSURED SOBRE UM WEB SERVICE DE UMA EMPRESA REAL. 72 f. Trabalho de Conclusão de Curso – Curso de Ciência da Computação, Universidade Tecnológica Federal do Paraná. Medianeira, 2016.

A velocidade com que o mercado evolui induz empresas a desenvolverem produtos no menor tempo possível, forçando as equipes de desenvolvimento a trabalharem contra o relógio. Os funcionários, pressionados para entregar os produtos dentro do prazo estabelecido, removem a etapa de teste do ciclo de desenvolvimento de software. As consequências deste corte passam a ser visíveis nas últimas fases de desenvolvimento quando problemas como falhas de segurança e comunicação entre módulos, começam a surgir. Como consequência, produtos são inseridos no mercado com péssima qualidade. Logo, testes deixam de ser opcionais e passam a ter espaço fixo dentro das instituições, sendo aplicados manualmente pelos desenvolvedores. Porém, isso demanda tempo e frequentemente não cobrem todo o sistema. Além disso, profissionais encarregados desta tarefa precisam montar todos os casos e fazer a análise dos resultados posteriormente, tirando-os do foco principal que seria a supervisão dos mesmos. Em vista disso, sistemas automatizados de teste passam a ser empregados na instituição. Estes sistemas são reutilizáveis, fáceis de atualizar, possuem baixos custos de implantação e são também muitos mais eficientes do que os testes manuais, além de trazerem confiança e consistência ao sistema. O sistema sobre teste tem como principais objetivos na divulgação de eventos e a venda de ingressos para eventos de um modo fácil e rápido, bem como todos os testes foram aplicados sobre uma imagem do sistema rodando em um servidor local. Este trabalho tem por objetivo aplicar o *framework* de teste REST Assured sobre este sistema, utilizando testes de unidade construídos por técnicas de *script* modular, verificando a aplicabilidade do *framework* para este tipo de sistema. Hipóteses foram estabelecidas junto aos membros da equipe da empresa com base nos requisitos de *Quality of Service* selecionados: desempenho, integridade e segurança. Alguns serviços passaram nos testes criados, bem como diversos serviços falharam, apontando erros envolvendo funções essenciais do sistema. Todos os resultados coletados foram analisados e repassados para a empresa. O *framework* mostrou-se uma boa solução no desenvolvimento e aplicação de testes em *Web services RESTful*.

Palavras-chave: teste de software, Web service, REST Assured

ABSTRACT

TREVISAN, Thiago B. APPLICABILITY OF FRAMEWORK REST ASSURED ON A COMPANY'S WEB SERVICE. 72 f. Trabalho de Conclusão de Curso – Curso de Ciência da Computação, Universidade Tecnológica Federal do Paraná. Medianeira, 2016.

Everything is happening so fast in the market and companies need to develop products in shorter times, pressuring the development teams to work against the clock. The employees, under pressure to deliver final products before the deadline, remove the testing phase of the software development cycle. The consequences start to arise in the last phases of the development, as security flaws and lack of communication between modules. As a result, low quality products are inserted in the market. So, tests are not an option any more and are established permanently inside institutions, being applied manually by the developers. However, applying and developing test cases manually takes time and frequently does not cover all the system. Besides, employees in charge need to mount all the cases and analyse the results after, putting them away from the main focus that would be the supervision of the tests. In view of this, automated testing systems are inserted into the institution. This systems are reusable, easy to update, have low deployment costs and are also more efficient than manual testing, besides bringing trust and consistency to the system. The system under test aims event advertising and a quick and easy ticket selling, as well as all the tests were applied on a system image running on a local server. This paper aims to apply the testing framework REST Assured on this system, using unit testing built by techniques of modular script, verifying its applicability in this type of system. Hypotheses were established within the company's members based on Quality of Service requirements: performance, integrity and security. Some services passed on the tests, as well as several have failed, finding errors even in essential system functionalities. All the collected results were analysed and forwarded to the company. The framework proved to be a good solution for developing and applying tests in Web services RESTful.

Keywords: software testing, Web service, REST Assured

Aos meus pais, Airton e Jandira.

Com organização e tempo, acha-se o segredo de fazer tudo e bem feito.
(Pitágoras)

AGRADECIMENTOS

A Deus por ter me dado saúde e força.

Aos meus pais, pelo amor, apoio e incentivo incondicional.

Ao meu orientador, Marcio Angelo Matté, pelo apoio, orientação, incentivo e confiança.

E a todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE SIGLAS

.CSV	*Comma-Separated Values*
API	*Application Programming Interface*
ATDD	*Acceptance Test-Driven Development*
BDD	*Behaviour-Driven Development*
DAO	*Data Access Object*
DSL	*Domain-Specific Language*
EATDD	*Executable Acceptance Test-Driven Development*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
JARs	*Java Archive*
JSON	*JavaScript Object Notation*
JVM	*Java Virtual Machine*
MVC	*Model View Controller*
ODBC	*Open Database Connectivity*
OOP	*Object-Oriented Programming*
POM	*Project Object Model*
QoS	*Quality of Service*
REST	*Representational State Transfer*
ROA	*Resource-Oriented Architecture*
RPC	*Remote Procedure Call*
SO	Sistema Operacional
SOAP	*Simple Object Access Protocol*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
TDD	*Test-Driven Development*
UDDI	*Universal Description, Discovery and Integration*
URI	*Universal Resource Identifier*
W3C	*World Wide Web Consortium*
WSDL	*Web Service Description Language*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Repetir o trabalho várias vezes é muito entediante quando feito manualmente e leva pessoas a cometerem equívocos, como escrever a mesma entrada de dados diversas vezes ou checar novamente linhas de código. Além do fato de que ao executar testes manualmente, tende-se a repetir os mesmos testes ocasionando o que chamam de 'The pesticide paradox'[1], onde testes executados repetidamente irão localizar cada vez menos problemas. A documentação necessária cresce em proporção ao número de testes realizados, então quanto mais testes, maior serão os gastos (HENDRICKSON, 2006). Quando da necessidade de reescrever algumas partes do código, pessoas tendem a fazer minúsculas modificações, imperceptíveis no momento, mas que não respondem da mesma forma do código anterior gerando inconsistência (TOMB; FLANAGAN, 2012) e repetibilidade (PHAM et al., 2013). Outro caso de consistência muito comum e difícil de se detectar é quando um ponto do programa se baseia numa hipótese para executar os processamentos e um segundo ponto deste mesmo programa utiliza de outra hipótese e estas duas suposições são de alguma forma incompatíveis (TOMB; FLANAGAN, 2012).

Pessoas tentam avaliar valores retornados pelo software de forma imparcial, mas sem perceber adicionam características pessoais ou julgamentos preconcebidos e convicções que as levam a diagnosticar de forma errônea os dados reduzindo o controle sobre o software e falta de informações sobre o comportamento do mesmo. Erros ocorridos durante o desenvolvimento do software normalmente são apresentados pelo IDE (do inglês, *Integrated Development Environment*) utilizado, na forma de linhas corridas apresentando a linha onde ocorreu erro na compilação e a mensagem de erro retornada do sistema. Estas informações são difíceis de serem interpretadas e entendidas pela mente humana.

Testes frenquentemente são evitados durante as primeiras fases do ciclo de desenvolvimento de software levando ao surgimento de erros durante as etapas finais. Estes erros podem ainda não ser despercebidos e serem mantidos na versão final do software. Estes erros serão convertidos em horas extras de serviço, adicionais de pagamentos aos funcionários, custos de redesenho, horas e horas de atendimento e suporte ao cliente, tirando o foco da

[1] Pesticide Paradox: "Todo método utilizado para prevenir ou localizar falhas deixa um resíduo de sutis falhas que os métodos serão ineficazes." (BEIZER, 1990)

empresa que no momento da venda de seu produto deveria ser exclusivamente no próprio negócio. Estes mesmos problemas ocorrem baseados no princípio de que softwares são escritos por humanos, e como tais, imperfeitos, cometem equívocos. Estes equívocos quando executados geram falhas (ou defeitos) que se não tratadas irão para a versão final do produto, resultando em perda da confiança do cliente sobre o produto e a empresa, perda de fatia do mercado e em situações extremas, término da vida útil do produto (MBURUGU, 2012; KEARNEY; RANDS, 2015).

Falhas no design de páginas Web levam pessoas, possíveis futuros clientes, a abandonarem a página no momento em que a acessam a página inicial, seja pela página demorar muito para carregar ou perceberem que o site não possui as informações que procuram, pela página necessitar de Flash[2] ou uma música começar a tocar no momento em que a página é carregada (ANTE MERIDIEM DESIGN, 2015). Páginas Web são consideradas cartões de visita, representam a imagem da empresa por si só, sua identidade. Experiências insatisfatórias de usuários durante a navegação refletirá diretamente na opinião deles sobre a companhia. Exemplos seriam erros de gramática, links quebrados, dificuldade em localizar informações, estrutura da página, entre outros (MBURUGU, 2012).

Teste de Software pode ser definido como conduzir uma verificação e validação do produto de software (SATHISH, 2012). É descrito como o processo de avaliar um sistema ou seus componentes com o intuito de achar algo que satisfaça os requisitos ou não. Testar um software é identificar quaisquer falhas, equívocos ou requisitos faltantes que vão contra os requisitos preestabelecidos. Outros autores descrevem teste de software como uma investigação conduzida com o intuito de prover aos *stakeholders* informações sobre a qualidade do produto de software ou serviço que está sobre teste. Mas, resumindo, teste de software tem três propósitos por objetivo, verificar, validar e localizar falhas na aplicação (BENTLEY; BANK, 2005).

Em testes de software automatizados, profissionais criam cenários de teste, desenvolvem códigos que executam estes testes e fazem a análise, se preocupando somente com a supervisão dos mesmos, ao contrário dos testes manuais, onde profissionais executam toda a tarefa. Estes testes são criados baseados no ambiente do usuário comum (CASLINO, 2014). Sistemas automatizados são reutilizáveis, e em cada nova falha ou novo teste descoberto, o sistema pode ser atualizado sem a necessidade de reentrar todos os dados utilizados nos testes, gerando um menor custo de implantação. São também mais rápidos para executar testes comparando com testes desempenhados por seres humanos, além de trazer consistência e confiança ao sistema reduzindo margens de erro em cenários de teste por meio de instruções

[2]http://www.adobe.com/products/flashplayer.html

já armazenadas e que não terão a possibilidade de serem esquecidas ou ignoradas pelo testador (WILLIAMS, 2004; SELLEO, 2015).

Teste de aplicações Web é considerada uma variação do teste de software convencional focado e exclusivamente adotado para testar aplicações hospedadas em ambiente Web. Testes normalmente ocorrem nas interfaces e em outras funcionalidades. Estas aplicações Web são considerados por alguns autores como toda aplicação que pode ser acessada por meio de um navegador (PALANI, 2011). Outros autores consideram testes de aplicações Web como a execução da aplicação utilizando várias combinações de entrada com o intuito de revelar falhas, sendo estas causadas talvez por problemas no ambiente de execução ou mesmo na própria aplicação. Metodologias comuns são teste de usabilidade, teste funcional, teste de desempenho, teste de aceitação do usuário, teste de segurança e teste de interfaces (GARCÍA; DUEÑAS, 2011; HALFOND et al., 2009).

O REST (do inglês, *Representational State Transfer*) é um estilo arquitetural para sistemas *hypermedia* distribuídos. REST é um estilo híbrido derivado de vários outros estilos arquiteturais baseados em rede combinado com restrições adicionais que definem uma interface de conexão uniforme. A ideia por trás do estilo pode ser descrita como um estilo arquitetural composto por um conjunto de restrições aplicados sobre os elementos da própria arquitetura. Após examinados os efeitos das restrições pode-se identificar as propriedades induzidas pela arquitetura Web. Restrições adicionais podem então ser incorporadas a arquitetura para formar um estilo que melhor reflete as propriedades desejadas de uma arquitetura moderna de Web (FIELDING, 2000). REST descreve a Web como uma aplicação a qual seus recursos se comunicam trocando representações de estado de recurso (COWAN, 2005; WEBBER et al., 2010).

1.1 OBJETIVO GERAL

Aplicar testes utilizando o *framework* REST-Assured sobre um *Web service* de uma empresa real, desenvolvido em Node.js, por meio de testes de unidade, construídos por técnicas de *script* modular com o intuito de verificar a aplicabilidade do *framework* para o sistema.

1.2 OBJETIVOS ESPECÍFICOS

Esse objetivo principal pode ser dividido nos seguintes objetivos específicos:

- Levantar referencial bibliográfico;
- Selecionar quais qualidades de serviço serão abordadas com base no sistema;
- Levantar hipóteses para cada uma das qualidades;
- Construir *Mock Objects*[3] de dados para teste;
- Aplicar os testes utilizando a ferramenta;
- Coletar e apresentar os resultados;
- Gerar considerações sobre a aplicabilidade do *framework*.

1.3 JUSTIFICATIVA

O mercado está muito competitivo com datas de entrega do produto cada vez mais apertadas, pressão sobre a empresa em entregar um produto de qualidade crescendo constantemente (HAYES, 2004). Produtos com atraso geram menores lucros, perda de clientes e de fatia do mercado, mas enquanto é ruim para a empresa entregar produtos com atraso, é péssimo inserir no mercado um produto com defeito (GRATER, 2005). Com isso, testes deixaram de ser opcionais e passaram a ser parte do ciclo de desenvolvimento do software. Para entregar um produto final de qualidade para o cliente, com o desenvolvimento de sistemas cada vez mais rápido e facilitado, o número de testes necessários também cresceu. Desempenhar estes testes manualmente é uma possibilidade de sanar este problema, mas necessita de um número considerável de profissionais, os quais como seres humanos, podem cometer equívocos ou não perceber falhas no sistema, acarretando em horas extras de serviço, custos de redesenho e clientes insatisfeitos (LIJUAN et al., 2012; SRIVASTAVA; KIM, 2009).

Este trabalho estudará e analisará a área de teste de software relacionado a aplicações Web focando nas ferramentas e *frameworks* de teste de software existentes, apresentando o *framework* REST-Assured e descrevendo suas características. Além disso, tem por objetivo desenvolver e aplicar testes sobre um *Web service* de uma empresa real com o intuito de testar

[3] Objetos que imitam objetos reais para teste

a ferramenta e verificar sua aplicabilidade neste tipo de serviço.

1.4 ESTRUTURA DO TRABALHO

Esse documento será organizado da seguinte forma. O Capítulo 2 apresentará o referencial teórico com as descrições de teste de software, suas fases e técnicas, e *Web services*, e suas tecnologias, bem como exibir padrões de qualidade de serviços e técnicas de desenvolvimento. Em seguida, REST é apresentado, descrevendo sua estrutura e propriedades, do mesmo modo que *frameworks* e ferramentas de teste são descritas. Por fim, apresentando o *framework* REST-Assured. A metodologia utilizada se encontra no Capítulo 3. Nele são descritas todas as etapas para o desenvolvimento do trabalho incluindo conceitos de algumas ferramentas que auxiliaram no desenvolvimento dos testes, os métodos de aplicação, as hipóteses levantadas e os experimentos realizados. No Capítulo 4 encontra-se os resultados obtidos durantes os testes, e por fim, as considerações finais são apresentadas no Capítulo 5.

2 LEVANTAMENTO BIBLIOGRÁFICO

O referencial bibliográfico para este trabalho é apresentado a seguir. Primeiramente, apresenta-se o conceito de teste de software, apresentando suas técnicas e fases na seção 2.1 e paradigma de desenvolvimento dirigido por teste na seção 2.2. Definições de *framework* para automatização de testes são apresentadas logo após, na seção 2.3, e *Web services*, exibindo suas características e principais tecnologias associadas na seção 2.4. Qualidade de serviço (QoS) é descrita na seção 2.5, apontando algumas das normativas e modelos adotados para determinar a qualidade do serviço, e da arquitetura REST, na seção 2.6, descrevendo seu funcionamento e sua relação com a arquitetura orientada à recursos ROA (do inglês, *Resource-Oriented Architecture*), apresentando conceitos de recursos e URI (do inglês, *Universal Resource Identifier*), e como eles se comunicam. Por fim, a ferramenta utilizada neste trabalho, o *framework* REST Assured, na seção 2.7.

2.1 TESTE DE SOFTWARE

Teste de software é o processo de gerar casos de teste para executar em um determinado programa com o intuito de analisar os resultados do teste e revelar possíveis falhas, erros e equívocos neste sistema. Um caso de teste é um conjunto de entradas para o programa que possui condições para a execução e que fornece uma determinada saída, denominada ”resultados esperados”. São normalmente criados para satisfazer um critério ou uma adequação e então usados para executar a aplicação e obter os resultados. Já um pacote de testes é uma coleção destes casos de teste, e rodar o programa com base no pacote de testes apropriado é a execução do teste. Para verificar a exatidão do sistema, são comparados valores de saída dos resultados esperados e dos resultados obtidos (SAMPATH, 2006; RUTH, 2007). O teste de software segundo Pressman (2010), é um elemento essencial no desenvolvimento do sistema, garantindo que o produto se enquadre nos padrões de qualidade de software, podendo consumir

em até 40% do esforço expedido pela equipe, uma vez que um defeito encontrado em fases posteriores do desenvolvimento gera um gasto maior para ser corrigido, tanto de tempo como financeiramente (SOLINO, 2008). Os testes são caracterizados de três formas, técnica estrutural ou caixa branca, técnica funcional ou caixa preta e técnica de teste baseado em erros.

Na caixa branca, o programador analisa a estrutura interna do software para estabelecer critérios e requisitos do sistema e depende de suas habilidades para identificar todos os caminhos dentro do software. Após identificá-los, o próximo passo é escolher os casos de teste que irão percorrer o código e determinar as saídas esperadas. Alguns dos critérios utilizados nesta técnica são: baseados em complexidade, baseados em fluxo de controle e baseados em fluxo de dados. Já na caixa preta, o programador, a partir dos critérios e requisitos estabelecidos por meio das funções do software pré-estabelecidos na especificação, testa os requisitos funcionais do software e determina as entradas válidas e inválidas sem nenhum conhecimento da estrutura interna do objeto a ser testado. Ambos possuem suas vantagens, o primeiro consegue cobrir um número considerado de casos de teste e o segundo cobre partes ainda não implementadas do sistema (RUTH, 2007; ÁRIAS, 2011). Por outro lado, a técnica de teste baseado em erros foca nos defeitos, erros conhecidos e comuns para derivar os requisitos dos casos de teste, bem como erros típicos cometidos durante o processo de desenvolvimento de software (SOLINO, 2008; ÁRIAS, 2011). Além disso, estão divididos em fases, descritas a seguir, que representam possíveis grupos existentes dentro do sistema para teste, desde testes individuais até testes que envolvam o sistema inteiro.

2.1.1 Fases de teste

O teste de unidade é um procedimento utilizado para validação das unidades individuais de um sistema, como um método ou uma classe da aplicação, considerando o paradigma orientado a objeto, verificando se elas estão funcionando de acordo com o esperado e procurando identificar erros de lógica e de implementação isolados em cada módulo. Uma unidade é considerada a menor parte passível de teste em uma aplicação. Tem por principal objetivo isolar cada parte do programa e mostrar se elas estão corretas. Cada teste de unidade gera um acordo, um documento escrito a qual o código deve satisfazer. Este tipo de teste suporta mudanças no sistema, sendo fácil testar quaisquer modificações ocorridas, e simplifica a integração devido as próprias unidades eliminarem a incerteza do sistema. Uma de suas

desvantagens é não ser capaz de capturar todas as falhas do sistema, pois por definição só cobre as unidades individualmente. Recomenda-se utilizar este teste em conjunto com outros testes, como o de integração (RUTH, 2007; SAMPATH, 2006; ÁRIAS, 2011). Em termos de *Web services*, as operações pertencentes a um serviço pode ser consideradas as unidades a serem testadas (BOZKURT, 2013).

Teste de integração, por sua vez, combina estes módulos analisados no teste de unidade e testa-os em conjunto e tem por objetivo verificar se atendem os requisitos de confiabilidade, desempenho e funcionalidade pré-estabelecidos que compõem o sistema. Considerando ainda *Web services*, isto implica no teste de sistemas compostos onde são verificados as interações entre seus serviços. Além disso, todo bloco formado já verificado passa para uma base de dados que será usada posteriormente para prover suporte para os próximos blocos formados. No entanto, não é possível analisar condições fora do escopo do teste que não estejam inclusas no caminho de execução dos itens da especificação, por exemplo, modificações que abrangem grande parte do sistema (RUTH, 2007; SAMPATH, 2006). Já Árias (2011), utiliza este teste como forma de descobrir erros associados às interfaces entre módulos, verificando se a estrutura gerada por meio de sua integração está de acordo com o modelo determinado em projeto.

Por outro lado, Bozkurt (2013) considera o teste de integração crucial para a garantia de que todos os componentes de um sistema funcionem corretamente quando trabalhados em conjunto. E seguindo a ideia de ROA, descrito na seção 2.6, teste de integração é tido como importante pois permite que todos os elementos de um sistema orientado à recursos possam ser testados, como os próprios serviços, mensagens, interfaces e os componentes em geral.

Considerando o teste de sistema como a fase final do teste de integração, onde o sistema é testado não só verificando se as partes trabalham em conjunto, mas também se o sistema como um todo funciona corretamente. Este teste tem por objetivo determinar se os componentes do sistema trabalham corretamente em conjunto com o intuito de atingir um algum propósito, implicando na verificação do sistema quanto ao cumprimentos dos requisitos funcionais e não-funcionais preestabelecidos (RUTH, 2007).

2.2 PARADIGMA DE DESENVOLVIMENTO DIRIGIDO POR TESTE

O paradigma dirigido por teste tem por princípio a escrita de testes antes da implementação do código de produção. Este paradigma inverte o fluxo de trabalho dentro do ciclo de desenvolvimento o qual antes envolvia o design, implementação, validação e

verificação das atividades, entre outros (CHELIMSKY et al., 2010; MäKINEN, 2012).

Nesta seção, apresenta-se algumas das principais práticas de desenvolvimento dirigido por teste, começando pelo *test-driven development*, fundamento essencial para todos os outros métodos, o qual trabalha com o desenvolvimento de testes automatizados escritos pelo próprio desenvolvedor na criação de aplicações de software (BECK, 2003). *Acceptance test-driven development*, por sua vez, segue a mesma ideia, mas é focado nos testes preparados em colaboração com os *stakeholders* (KOSKELA, 2007). Por último, o *behaviour test-driven development* que organiza os problemas em cenários, e propicia uma melhor visualização das interações entre pessoas e sistemas, e seus comportamentos (CHELIMSKY et al., 2010).

2.2.1 *Test-Driven Development*

Segundo George (2002) e Hilton (2009), *test-driven development* (TDD) é considerado uma das principais práticas de *eXtreme Programming* (XP) e é utilizado para design e desenvolvimento de código de forma incremental, garantindo que somente são escritos os códigos necessários para o funcionamento do sistema e que todo código seja inerentemente testado.

Já Dohmke (2008) menciona que o TDD possui duas regras básicas. Primeiro, um novo código é escrito somente se um teste automatizado falhou, e segundo, códigos duplicados são eliminados, formando assim o seguinte ciclo de desenvolvimento:

- Escrever o código de teste;
- Rodar o teste, verificar se o mesmo falha;
- Rodar o código novamente fazendo as mudanças neccessárias para o teste passar;
- Remover os códigos duplicados.

Por outro lado, Beck (2003), Jasek (2014) e Nilsson (2015) citam que TDD é um processo de desenvolvimento de software que apoia-se na ideia de repetição de um ciclo de desenvolvimento, onde testes são escritos antes dos códigos de produção. Essa repetição ocorre até que se obtenha testes para todas as funcionalidades requeridas pelo sistema e que o código passe em todos os testes. Primeiramente, a tarefa do desenvolvedor é escrever um teste que represente uma nova funcionalidade do sistema, garantindo que este teste irá falhar pois esta funcionalidade ainda não foi implementada. Ainda, dá ênfase ao fato de que um único teste unitário é o suficiente para falhar e que nunca deve-se trabalhar com mais de um teste falho ao

mesmo tempo. Segundo, deve-se escrever código de produção. O detalhe desta fase é que deve-se escrever somente código suficientemente necessário para passar no teste, mesmo sabendo que este código poderá mudar no futuro, fato que desenvolvedores apontam como uma das desvantagens da técnica. Por último, vem a etapa de refatoração do código. O objetivo desta etapa é remover códigos duplicados, aumentar a clareza e expressividade do código, bem como reduzir o acoplamento e aumentar a coesão do código, não somente dos códigos de teste e produção, mas também todo e qualquer código que precise de refatoração. Fala ainda que este ciclo pode ser representado na forma de três leis:

- Não deve-se escrever código de produção antes de ter escrito um código de teste de unidade que gere uma falha;
- Deve-se escrever um código de teste suficientemente para que o teste falhe;
- Deve-se escrever somente o código de produção necessário para passar no teste.

No desejo de se trabalhar o TDD juntamente com os *stakeholders* é apresentado uma outra abordagem do método denominada *acceptance test-driven development*.

2.2.2 *Acceptance Test-Driven Development*

Conforme Koskela (2007) e Koudelia (2011), o *acceptance test-driven development* (ATDD) é uma técnica de desenvolvimento de software que combina a especificação dos requisitos e o desenvolvimento de testes automatizados para aplicá-los sobre os requisitos, e possui três principais características. Primeiro, ele fornece um meio de comunicação compartilhada para a troca de informações entre os *stakeholders*, como testadores, desenvolvedores, e especialistas do domínio (pessoas que fornecem os requisitos do software), propiciando não só um melhor fluxo de informações entre estes grupos, bem como um melhor fluxo dentro de cada um deles. Segundo, ATDD fornece instrumentos para armazenamento de documentação de software, mantendo-a atualizada durante toda a fase de desenvolvimento. Por fim, ATDD se certifica de que o sistema esteja sempre de acordo com os requisitos por meio de testes automatizados e mantenha um código visualmente agradável por meio de constantes refatorações.

De acordo com Melnik (2007), ATDD é abordada como *executable acceptance test-driven development* (EATDD), técnica de desenvolvimento que torna possível formalizar as expectativas da empresa em especificações passíveis de leitura e execução, as quais possibilitam

que desenvolvedores consigam produzir e finalizar um sistema. Além disso, Khandkar et al. (2009) menciona que EATDD baseia-se na perspectiva dos *stakeholders* com o intuito de auxiliar desenvolvedores a obter um melhor entendimento dos requisitos do sistema e validá-los sobre estes requisitos. Ainda, EATDD segue a ideia do TDD e determina que para a adição de uma funcionalidade, é necessário antes existir um teste de aceitação para o mesmo. Estes testes de aceitação podem ser acessados, revisados e executados por qualquer membro da equipe o que proporciona um maior grau de confiança sobre o sistema que está sendo desenvolvido.

2.2.3 *Behaviour-Driven Development*

Segundo NEČAS (2011), o *behaviour-driven development* (BDD) surge para solucionar algumas falhas do TDD. Possui um vocabulário diferente, onde substitui as palavras *unit* e *test* por *behaviour* e *specification*. Esta mudança no vocabulário leva ao desenvolvimento de especificações no lugar de testes, levando o foco dos testes para o que o software deveria fazer ao invés de verificar se o software faz aquilo ou não. Aponta ainda alguns princípios que servem de base para o desenvolvimento:

- somente o necessário: somente utilizar o esforço necessário para gerar um software de valor para o cliente;
- entregar valor aos *stakeholders*: *stakeholders* não só representado aqui como o cliente, mas também todo consumidor que obterá um ganho utilizando-o;
- tudo envolve comportamento: desde os detalhes de implementação até o alto nível, tudo descreve comportamento.

Conforme NEČAS (2011) e Chelimsky et al. (2010), BDD foca na implementação da aplicação baseado em seu comportamento definido juntamente com os *stakeholders*. Além disso, Bołoz (2014) e Delshad (2016) mencionam que BDD segue os princípios do ATDD no intuito de ajudar os membros da equipe a entender quais são as necessidades dos *stakeholders*, apresentando os dados na linguagem do consumidor. Dessa forma, os testes conseguem ser pequenos e específicos, normalmente testando um único requisito. Essa linguagem facilitada não só ajuda os *stakeholders* a expor e especificar os testes necessários, como também cria uma transparência entre as expectativas do cliente e os testes gerados pelos desenvolvedores, levando a uma otimização do processo (FOFUNG, 2015).

Por outro lado, Vance e Cickovski (2012) reforça que BDD segue os princípios do TDD e adiciona que o requisito principal do método é a construção de testes semânticos que

tem por definição a utilização da palavra *should* (do inglês, deveria).

Por último, Chelimsky et al. (2010) apresenta a divisão dos requisitos provenientes dos *stakeholders* em diferentes componentes do BDD: *features*, *stories*, *scenarios* e *steps*. *Features* são os requisitos funcionais que formam o padrão de design do sistema dirigido por comportamento. Uma *feature* pode possuir diversas *stories*, que realçam o inter-relacionamento de *stakeholders* e *features* e podem ser apresentadas por formatos baseados em *template*. Uma *story*, pode ser composta de diversos *scenarios*, os quais exibem como determinadas situações ocorrem, destacando que *positive scenarios* descrevem interações que obtiveram sucesso enquanto que *negative scenarios* demonstram condições particulares que geraram uma exceção. Por fim, cada *scenario* contém um ou mais *steps*, os quais compõem os menores indicadores de comportamento.

Alguns dos mecanismos utilizados para a aplicação destes paradigmas é abordado da próxima seção falando dos *frameworks* e ferramentas de teste automatizado.

2.3 FRAMEWORKS E FERRAMENTAS DE TESTE

Tecnologias de automação melhoram a cobertura dos testes sobre o software e geram produtos de maior qualidade. Salvam milhares de horas em execução de testes manuais, reduzindo assim os custos da empresa. Existem vários artifícios sobre os quais é possível aplicar testes em softwares, e os meios abordados neste trabalho serão *frameworks* e ferramentas de teste automatizado.

2.3.1 Ferramentas de teste

As ferramentas de teste de software são usadas como parte da fase de testes do ciclo de desenvolvimento de software para automatizar certas tarefas, melhorar a eficiência dos testes e descobrir falhas que seriam muito difíceis de serem encontradas usando testes manuais. A análise do código pode ser feita de duas formas, estática e dinâmica (DUPAUL, 2015).

Analisar o código de forma estática consiste em ler o código do programa usando

técnicas de execução simbólica para simular execuções abstratas do programa com o intuito de computar entradas que direcionam o software para determinado caminho sem sequer executar o programa. Infelizmente, esta abordagem está limitada a situação onde o programa possui declarações que envolvem restrições que estão fora do escopo de raciocínio, como manipulações de ponteiro, operações aritméticas, entre outros. E testes dinâmicos apoiam-se na ideia de executar um programa começando por inserir alguns valores de entrada aleatórios coletando informações de restrições simbólicas nas entradas de declarações condicionais ao longo da execução, usando um solucionador destas restrições para inferir novos valores de entrada com o intuito de deslocar a próxima execução do programa para uma outra direção. O processo é repetido até que uma sentença específica seja alcançada (GODEFROID et al., 2008).

Estas ferramentas muitas vezes são superestimadas, elas dizem ser capazes de executar algumas tarefas, no entanto, na prática falham terrivelmente. Funções como RECORD e PLAYBACK, por exemplo, tem um desempenho fraco quando executados os testes. Funcionam perfeitamente em pequenos projetos mas são sistemas que não possuem escalabilidade e geralmente não são portáveis e difíceis de serem customizados. A partir disso, surge o *framework* de teste.

2.3.2 Frameworks de teste

Os *frameworks* de teste, que são ambientes de execução para testes automatizados. É definido como um conjunto de hipóteses, conceitos e práticas que constituem uma plataforma de trabalho ou suporte a testes automatizados, responsáveis por definir o formato no qual serão exibidos os resultados esperados, criar um mecanismo para guiar a aplicação durante o teste, executar estes testes e apresentar os resultados obtidos. É mais fácil de trabalhar do que as ferramentas de teste e simples de ampliar a estrutura para projetos maiores ou expansão do próprio projeto atual. Normalmente trabalha em módulos facilitando a adaptação em caso de alterações no projeto, e possui custos de manutenção reduzidos. Eles são divididos em 4 categorias: *script* modular, orientado por dados, orientado por palavra-chave e híbrido (LAUKKANEN, 2006).

O teste modular consiste na criação de pequenos *scripts*, independentes entre si, que representam módulos, seções e funções da aplicação que está sobre teste atualmente. Podem ser agrupados de forma hierárquica para desempenhar testes em larga escala. Estratégia de

programação bem conhecida, considerada a de mais fácil aprendizado, foca na construção de uma camada abstrata em frente ao componente para esconder este componente do resto da aplicação, isolando assim a aplicação das modificações pelas quais o componente irá passar, gerando modularidade no design da aplicação. Este conceito de abstração atrelado ao encapsulamento traz manutenibilidade e escalabilidade aos pacotes de testes automatizados. Uma das desvantagens do método surge quando da necessidade de atualização da base de dados do teste, onde alterações no código do *script* serão necessárias e isto pode ser um problema se o código for extenso, introduzindo assim o método orientado por dados (GHANAKOTA, 2012; KELLY, 2003).

Neste modelo, os valores de entrada e saída dos testes são lidos a partir de arquivos, como *datapools*, fontes ODBC, .CSV, Excel, objetos DAO, entre outros, e são atribuídos a variáveis usando *scripts*, manualmente ou não, exibindo como saída o status do teste em forma de arquivo *logging*. Isso permite que um único *script* possa rodar quase todos os casos de teste usando múltiplos conjuntos de dados. Possibilita redução do número de *scripts* necessários para executar todos os casos de teste, oferece flexibilidade durante a manutenção, principalmente na parte de desempenho, e conserto de bugs existentes e por último redução do código gerado para execução dos testes. No entanto, como os dados de teste e os *scripts* são fortemente relacionados, implica que quando ocorrem mudanças ou criação de novos casos de teste, é necessário a criação de novos *scripts* que compreendam estes novos dados. Com isso, passou a ser utilizado um novo método, orientado por palavra-chave (LAUKKANEN, 2006).

O *framework* dirigido por palavra-chave utiliza de tabelas de dados e palavras-chave e seu desenvolvimento independe da ferramenta de automação utilizada para executá-los, dos códigos que guiam a aplicação sobre teste e dos dados. Testes orientados por palavra-chave possuem uma grande similaridade com casos testes implementados manualmente. Por padrão, em um teste como este, a funcionalidade da aplicação que está sobre teste é documentada em uma tabela bem como as instruções passo a passo desempenhada por cada teste. Como possui todas as características do teste orientado por dados, as mesmas vantagens se aplicam para este método. Além disso, não necessita de um especialista em automação de testes para manter ou criar um novo conjunto de casos de teste e as palavras-chave podem ser reutilizadas por outros casos de teste. Entretanto, para desempenhar tais tarefas é necessário um *framework* mais complexo do que no dirigido por dados e por esta abordagem oferecer mais flexibilidade, os casos de teste ficaram maiores e complexos. Diante disso, foi combinado todos os modelos apresentados, agrupando suas características fortes e suavizando seus pontos fracos gerando o chamado *framework* de teste híbrido (KELLY, 2003).

Múltiplos projetos dentro da empresa e o próprio tempo levam o *framework* a evoluir

para o estilo híbrido. Ele permite que *scripts* dirigidos por dados tirem vantagem de poderosas bibliotecas e de funcionalidades da abordagem orientada por palavra-chave e ele não só consegue transformar estes *scripts* em modelos mais compactos e menos sensíveis a falhas, como também facilita a sua conversão para modelos equivalentes dirigidos por palavra-chave. Por outro lado, também, este framework consegue executar *scripts* para desempenhar tarefas geralmente difíceis de serem implementadas quando uma abordagem por palavra-chave é utilizada. Seu ponto negativo seria a alta complexidade existente para implantar o modelo (GRACIAS, 2010; GHANAKOTA, 2012; KELLY, 2003).

2.4 WEB SERVICES

Diversas definições são encontradas na literatura quando busca-se o conceito de *Web services*. Dentre elas, pode-se citar algumas.

Um *Web service* é considerado uma parte da lógica de negócios dentro da organização, disponível na Internet, e acessível por meio de protocolos padrões como o HTTP (do inglês, *Hypertext Transfer Protocol*). Seu uso pode ser tão simples como fazer login em um *website* quanto complexo como administrar a estrutura de negócios entre várias organizações (CHAPPELL; JEWELL, 2002).

Conforme Soroor e Tarokh (2006) e Keen et al. (2012), *Web services* são aplicações dinâmicas, independentes, modulares e distribuídas que podem ser descritas por mecanismos padrões, publicadas e localizadas usando registros padrões, ou invocadas por meio da rede de Internet com o intuito de criar produtos, processos e redes de suporte. Podem ser locais, distribuídas ou baseadas em Web. Além disso, são criadas sobre protocolos abertos como o TCP/IP (do inglês, *Transmission Control Protocol/Internet Protocol*), HTTP, HTML (do inglês, *HyperText Markup Language*) e XML (do inglês, *Extensible Markup Language*) e utilizam de tecnologias como SOAP (do inglês, *Simple Object Access Protocol*) para troca de mensagens e UDDI (do inglês, *Universal Description, Discovery and Integration*) e WSDL (do inglês, *Web Service Description Language*) para publicação.

A seguir algumas características de comportamento de *Web services* de acordo com Chappell e Jewell (2002):

- Baseado em XML: protocolos e tecnologias de *Web services* podem facilmente trocar informações usando XML como camada de representação de dados e também, eliminar qualquer conexão dependente de rede, sistema operacional ou plataforma a usando na

camada de transporte de dados;

- Livre de acoplamento: a interface de um *Web service* pode ser transformada sem comprometer a habilidade do cliente de usufruir do serviço, pois este não está amarrado ao sistema diretamente. O uso desta arquitetura possibilita que softwares possam ser facilmente gerenciados e permite uma simples integração entre diferentes sistemas;
- Granulado grosso: tecnologias orientado objeto tem por característica a criação de métodos pequenos, que executam somente algumas instruções. Durante o desenvolvimento de um programa, o processo comum é a criação de métodos pequenos que serão incorporados a um serviço maior, o qual será consumido pelo cliente ou outro serviço. A tecnologia de *Web services* possibilita uma forma de definir estes serviços para acessarem a quantidade certa necessária para a lógica de negócios;
- Habilidade de ser síncrono ou assíncrono: sincronicidade refere-se a ligação entre cliente e execução do serviço. Em invocações síncronas, o cliente bloqueia e aguarda o serviço completar suas operações antes de continuar, enquanto que em operações assíncronas, o cliente invoca um serviço e então executa outras operações. Esta capacidade de ser assíncrono é um fator essencial para se ter sistemas livres de acoplamento;
- Suporte a RPCs (do inglês, *Remote Procedure Call*): *Web services* permitem que clientes invoquem procedimentos, funções e métodos em objetos remotos via protocolo XML. Os procedimentos remotos expõem parâmetros de entrada e saída os quais um *Web service* deve prover suporte. Um *Web service* gera suporte a RPCs criando seus próprios serviços, equivalentes a aqueles de um componente tradicional;
- Suporte a troca de documentos: uma das vantagens do XML é a forma genérica com a qual se representa não só dados, mas também complexos documentos. Estes documentos podem ser simples como representar um endereço, e complexos como representar todo um registro. *Web services* provém suporte a troca transparente de documentos com o intuito de facilitar a integração dos negócios.

E por fim, Albreshne et al. (2009) descreve um *Web service* como uma interface que apresenta uma coleção de operações que são acessíveis por meio da rede de Internet utilizando de trocas padrões de mensagens em XML. Esta interface permite que os detalhes da implementação sejam ocultos para o serviço, possibilitando que sua utilização seja independente de hardware ou plataforma, e também independente de linguagem de programação. Com isso, permite-se e gera-se suporte para que aplicações baseadas em *Web services* tornem-se livres de acoplamento, orientadas por componente, e que sua implementação seja independente de tecnologia (KREGER, 2001).

As tecnologias que intermeiam o âmbito dos *Web services* são apresentadas a seguir.

2.4.1 Tecnologias

As tecnologias descritas a seguir formam os padrões internacionais que intermeiam a tecnologia de *Web services* atualmente. São eles o XML, SOAP, WSDL e UDDI.

XML é uma linguagem de marcação base para *Web services*. É tida como uma linguagem genérica utilizada para descrever qualquer conteúdo organizado de forma estruturada, separada da sua representação para qualquer dispositivo. Todos os elementos de *Web services* utilizam XML, como os *namespaces* e os esquemas XML (KEEN et al., 2012). Além disso, a *World Wide Web Consortium* (W3C) apresenta XML como uma classe de objetos providos de conteúdo denominados documentos XML e que descreve parcialmente o comportamento de programas que os processam. Documentos XML são compostos de unidades de armazenamento denominadas *entidades*, as quais contém dados analisados ou não. Os dados analisados são compostos de caracteres, onde uns formam os dados e outros formam a marcação. A marcação, por sua vez, tem por finalidade formar o layout do armazenamento bem como codificar a estrutura lógica do documento (W3C, 2006).

SOAP é o protocolo de mensagens utilizado por *Web services*. Ele é moldado para permitir que plataformas distribuídas separadamente possam interoperar, e alcança seus objetivos pela seguintes características: simplicidade, flexibilidade, mensagens baseadas em XML. SOAP utiliza de protocolos como o HTTP para acessar recursos Web. Sua função é definir como a mensagem é formatada, não tendo influência sobre o modo dela ser entregue (ALBRESHNE et al., 2009). De acordo com Chappell e Jewell (2002), SOAP oferece uma estrutura padrão para transporte de documentos XML por meio de diversos tecnologias como SMTP, HTTP e FTP.

Segundo a W3C (2007), SOAP é um protocolo voltado para a troca de informação estruturada em um ambiente distribuído e descentralizado. Ele utiliza tecnologias XML para definir um *framework* para troca de mensagens extensível com o intuito de prover uma construção de mensagem que pode ser transmitida por meio de diversos protocolos.

WSDL é uma linguagem XML desenvolvida com o intuito de representar um *Web service*. Considerando uma descrição WSDL como um contrato API com clientes, tem-se que a descrição especifica o endereço, mecanismos de comunicação, interface e os tipos de mensagens de um *Web service*. Enfim, representa toda a informação da qual o cliente precisa para utilizar um *Web service* (MANDEL, 2008).

A W3C (2001) descreve WSDL como

> um formato XML para descrever serviços de rede como um conjunto de *endpoints* operando por meio de mensagens contendo informações orientadas por procedimento ou documento. As operações e mensagens são descritas abstraticamente, e então são unidos à um protocolo de rede e formato de mensagem concreto com o intuito de definir um *endpoint*. *Endpoints* relacionados são combinados em *endpoints* abstratos (serviços). WSDL é extensível a fim de permitir a descrição de *endpoints* e suas mensagens independentemente do tipo de mensagem ou protocolo de rede são utilizados para comunicação.

Por outro lado, Chappell e Jewell (2002), tem WSDL como uma tecnologia XML que descreve a interface de um *Web service* de uma forma padrão. WSDL padroniza o modo com o qual um *Web service* representa os parâmetros de entrada e saída de uma invocação, a estrutura da função e a natureza da invocação.

Por fim, UDDI cria uma plataforma padrão que permite a troca de informações e permite que empresas e aplicações achem e usem, de forma dinâmica, fácil e rápido, os *Web services* por meio da Internet. Ele é um protocolo baseado em XML como SOAP e WSDL e oferece uma infraestrutura para endereçamento sistemático das necessidades dos *Web services*, como descoberta, gerenciabilidade e segurança (RAJENDRAN; BALASUBRAMANIE, 2009). Ainda, conforme Chappell e Jewell (2002), UDDI oferece um registro para *Web services* voltado à divulgação, descoberta e integração. Além disso, provém uma estrutura com o objetivo de representar empresas, seu relacionamentos e *Web services*. Surgiu no mercado com o intuito de fornecer uma abordagem para que empresas alcancem seus clientes e parceiros, apresentando seus produtos e *Web services*, e se tornar um método uniforme para a integração de sistemas e processos que já estão situados entre parcerias.

2.5 *QUALITY OF SERVICE*

Qualidade de serviço (QoS, sigla do inglês) refere-se às propriedades de *Web services* como desempenho, confiabilidade, disponibilidade, segurança, entre outros (FERNANDES; MEDEIROS, 2009).

Segundo W3C (2003), por outro lado, é apresentado como os requisitos que são referência para o aspecto qualitativo de um *Web service*.

Conforme a ISO/IEC 8402:1994 e apontado na tese de doutorado de Hilari (2015), QoS é a totalidade de funcionalidades e características de um produto ou serviço que se apoiam em suas habilidades para satisfazer as necessidades implícitas e explícitas, bem como um

modelo de QoS é a especificação das características necessárias que um sistema de software deve apresentar (ISO/IEC, 1994).

Por fim, Mani e Nagarajan (2002) cita que QoS cobre uma vasta gama de técnicas que combinam as necessidades dos requisitores de serviços com as dos provedores de serviços baseado em recursos de rede disponíveis, com o intuito de determinar a qualidade e usabilidade do serviço, o qual influencia em sua popularidade.

Diversos modelos de QoS podem ser encontrados na literatura e eles diferenciam entre si na terminologia utilizada, bem como pelo conjunto de características de qualidade que abordam e pela estrutura do modelo. Os modelos apontados neste trabalho são: a ISO/IEC 9126:2001 e sua sucessora, a ISO/IEC 25010-3:2011, a W3C e um modelo apontado por engenheiros de software da IBM, Mani e Nagarajan (2002) e a versão de Rajendran e Balasubramanie (2009).

2.5.1 ISO/IEC 9126:2001

A ISO 9126 define um modelo de qualidade interna e externa para produtos. Ela organiza os atributos de qualidade de software em seis categorias (funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade, portabilidade), as quais são subdivididas em características específicas que possuem influência sobre a característica de qualidade (ISO/IEC, 2001). A Figura 11 exemplifica o modelo. Definições para cada categoria e suas características específicas do software são apresentadas a seguir.

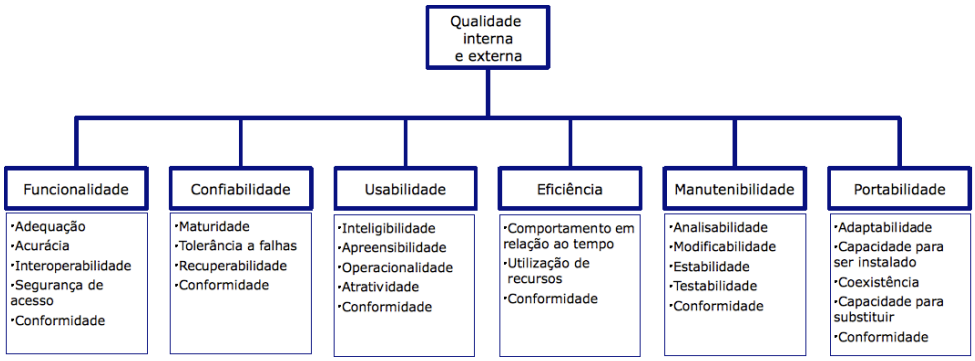

Figura 11 – Modelo de qualidade de produto ISO 9126
Wikipédia (2016)

a) Funcionalidade é a capacidade do produto de software de prover funções que satisfaçam

as necessidades estabelecidas quando da utilização sobre condições específicas. Suas características específicas são:

- adequação: capacidade de prover um conjunto de funções para tarefas específicas;
- acurácia: capacidade de prover resultados corretos necessários com um certo grau de precisão;
- interoperabilidade: capacidade de interagir com um ou mais sistemas;
- segurança: capacidade de proteger informações e dados para que pessoas não autorizadas possam lê-las ou modificá-las;
- conformidade: capacidade de se adequar a padrões, convenções ou normativas relacionados a funcionalidade.

b) Confiabilidade é a capacidade do produto de software em manter um nível específico de desempenho quando usado sobre determinadas condições. Suas características específicas são:

- maturidade: capacidade de evitar falhas em decorrência dos problemas no software;
- tolerância à falhas: capacidade de manter um nível de desempenho em casos de falhas de software ou infringimento de interfaces específicas;
- recuperabilidade: capacidade de restabelecer um nível específico de desempenho e recuperar os dados diretamente afetados em caso de falha;
- conformidade: capacidade de aderir à padrões, convenções ou normativas relacionados a confiabilidade.

c) Usabilidade é a capacidade do produto de software ser atrativo para o usuário, fácil de compreender, aprender e usar quando utilizado sobre condições específicas. Suas características específicas são:

- inteligibilidade: capacidade de permitir ao usuário compreender se o software é adequado e como pode ser usado em tarefas particulares;
- apreensibilidade: capacidade de fornecer uma rápida aprendizagem do software;
- operacionalidade: capacidade de permitir que usuário opere o software facilmente;
- atratividade: capacidade de ser atrativo para o usuário;
- conformidade: capacidade de se adequar à padrões, convenções e normas referentes a usabilidade.

d) Eficiência é a capacidade do produto de software fornecer o desempenho adequado de acordo com a quantidade de recursos utilizados, sobre as condições estabelecidas. Suas características específicas são:

- comportamento em relação ao tempo: capacidade de fornecer a reposta, tempos de processamento e taxas de transferência apropriadas quando da execução de funções;

- utilização de recursos: capacidade de utilizar recursos com quantidades e tipo adequados durante a execução de funções do software;
- conformidade: capacidade de aderir à padrões e normas pertinentes à eficiência.

e) Manutenibilidade é a capacidade do produto de software ser modificado. Modificações podendo incluir correções, aprimoramentos ou adaptações do software. Suas características específicas são:
   - analisabilidade: capacidade de diagnosticar deficiências ou causas de falhas no software, bem como partes a serem modificadas;
   - modificabilidade: capacidade de permitir a implementação de modificações especificadas;
   - estabilidade: capacidade de evitar efeitos inesperados oriundos das modificações do software;
   - testabilidade: capacidade de validação do software modificado;
   - conformidade: capacidade de estar em acordo com os padrões e convenções relacionados a manutenibilidade.

f) Por fim, portabilidade é a capacidade do produto de software ser transferido de um ambiente para outro. Suas características específicas são:
   - adaptabilidade: capacidade de ser adaptado para diferentes ambientes sem a necessidade de aplicar ações ou meios além dos requisitados para executar a ação;
   - capacidade de ser instalado: capacidade de ser instalado em um ambiente especificado;
   - coexistência: capacidade de coexistir juntamente com outros software independentes em um ambiente em comum, compartilhando dos mesmos recursos;
   - capacidade para substituir: capacidade de ser usado no lugar de outro software com o mesmo propósito e no mesmo ambiente;
   - conformidade: capacidade de se adequar a padrões, convenções ou normativas relacionados a portabilidade.

A ISO 25010 descrita a seguir apresenta uma revisão da normativa apontada pela ISO 9126 com o intuito de atualizar ou renovar os requisitos que um produto de qualidade deve apresentar.

2.5.2 ISO/IEC 25010-3:2011

A ISO 25010 tem por objetivo apresentar um modelo de qualidade voltado para a identificação de características de qualidade relevantes associadas aos objetivos e metas estabelecidos pelos *stakeholders* para o sistema, que poderão ser usadas posteriormente para estabelecer requisitos, seus critérios e medidas. Por meio deste modelo é possível formular uma especificação compreensiva e gerar uma avaliação da qualidade do software, essencial na geração de valor para os *stakeholders* (ISO/IEC, 2011). A Figura 12 exemplifica o modelo.

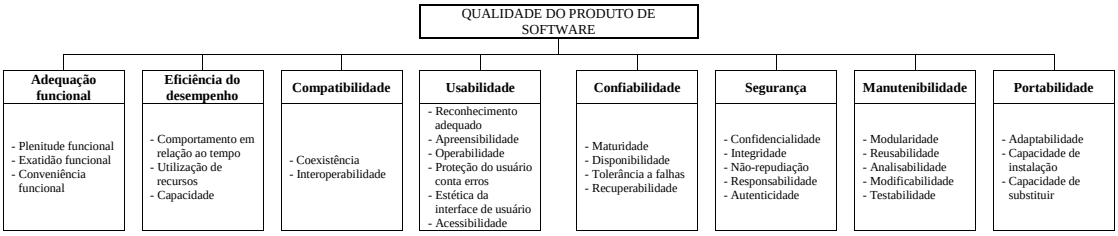

Figura 12 – Modelo de qualidade de produto ISO 25010

Por ser uma revisão da ISO 9126, compartilha das mesmas caraterísticas e propões algumas alterações:

- O escopo do modelo de qualidade foi estendido para incluir sistemas de computadores, e qualidade durante o uso na visão do sistema;
- *Segurança* foi adicionada como uma categoria ao invés de ser uma característica específica de *funcionalidade*;
- *Compatibilidade*, tendo *interoperabilidade e coexistência* como características, foi adicionada como categoria;
- *Plenitude funcional, capacidade, proteção do usuário contra erros, acessibilidade, disponibilidade, modularidade e reusabilidade* foram adicionadas como categorias específicas;
- A característica de *conformidade* foi removida por ser parte dos requisitos de sistema de modo geral.

Além disso, algumas características pertinentes ao trabalho são descritas a seguir:

- Confidencialidade descreve que um sistema deve garantir que os dados somente são

acessíveis à aqueles autorizados a ter acesso;

- Não-repudiação é a capacidade de ações ou eventos conseguirem provar que eles mesmos aconteceram, isto é, refere-se à incapacidade de um dos participantes de uma transação em mais tarde negar que tenha participado dela;
- Responsabilidade refere-se ao fato de que ações de uma determinada entidade podem ser rastreadas unicamente para aquela entidade;
- Autenticidade diz que a identidade de um sujeito ou recurso pode ser comprovada por quem o reivindicou.

O modelo da W3C e dos engenheiros da IBM, Mani e Nagarajan (2002), possui estrutura similar ao modelo abordado pelas ISOs, no entanto, apresenta um modelo de qualidade de produto com foco em *Web services*.

2.5.3 W3C e IBM

A W3C (2003) propõe aspectos de QoS de *Web services* incluindo acessibilidade, capacidade, confiabilidade, disponibilidade, escalabilidade, gerenciamento de exceções, integridade, interoperabilidade, desempenho, precisão, robusteza, segurança, requisitos de QoS relacionados à rede. Estes atributos são definidos a seguir:

- Acessibilidade: *Web services* devem fornecer alto grau de acessibilidade, no contexto de apresentar suporte as requisições dos clientes;
- Capacidade: número máximo de requisições simultâneas que um *Web service* consegue fornecer mantendo o mesmo desempenho;
- Confiabilidade: habilidade do *Web service* de executar suas funções sobre determinadas condições estabelecidas em um intervalo de tempo definido. É tida como a medida geral da capacidade de um *Web service* manter sua qualidade no serviço;
- Desempenho: representa o quão rápido uma requisição de serviço pode ser completada. Pode ser medida com base na taxa de transferência, tempo de resposta, latência, tempo de execução e transação, entre outros;
- Disponibilidade: é a probabilidade de um *Web service* estar funcionando, disponível para o uso. Possui relação direta com a confiabilidade;
- Escalabilidade: representa a habilidade de expandir a capacidade computacional de um sistema provedor de serviços e a habilidade do sistema de processar um número maior de

requisições de usuários. Possui relação direta com o desempenho;

- Gerenciamento de exceções: exceções, como em casos específicos e possibilidades não previstas, devem ser gerenciadas de forma adequada pelo fato de não ser possível prever todas as saídas possíveis do serviço;
- Integridade: capacidade do sistema ou componente de impedir e prevenir acesso não autorizado, ou modificação dos dados e programas do computador. São divididas em integridade dos dados e das transações;
- Interoperabilidade: *Web services* devem conseguir trocar informações entre si independentemente do ambiente de desenvolvimento onde foram implementados os serviços;
- Precisão: representa a taxa de erros apresentado pelo *Web service*, formulada pelo número de erros que o serviço gera sobre um determinado intervalo de tempo, a qual deve ser minimizada;
- Requisitos de QoS relacionados à rede: os mecanismos de QoS que controlam a aplicação devem trabalhar juntamente com os mecanismos de QoS que operam a camada de transporte da rede. Alguns parâmetros desta camada são o *delay* da rede, a variação do *delay*, a perda de pacotes, etc;
- Robusteza: representa a capacidade do *Web service* de funcionar corretamente na presença de valores de entrada inválidos, incompletos ou conflitantes;
- Segurança: o provedor de *Web services* deve aplicar diferentes políticas de segurança, variando o método de abordagem e níveis, com base no requisitor do serviço. Segurança para *Web services* significa prover autenticação, autorização, confidencialidade, rastreamento e auditorias, encriptação dos dados e não repudiação.

Mani e Nagarajan (2002) apontam as mesmas definições mencionadas acima e apresentam uma diferente proposta para escalabilidade e desempenho de *Web services*. Primeiro, escalabilidade passar a ser uma subcaracterística de acessibilidade e refere-se a habilidade de constantemente atender as requisições independentemente do volume de requisições. Por último, desempenho é baseado na taxa de transferência e latência, onde uma boa representação de um *Web service* é constituída de altas taxas de transferência e baixas latências. Além disso, adicionaram o requisito regulamentação a lista de atributos de QoS. Regulamentação menciona que um *Web service* deve estar em conformidade com as regras, leis e padrões estabelecidos, como SOAP, UDDI e WSDL, sempre atualizado com as últimas versões pelo fato de propiciar a chamada adequada de *Web services* pelos requisitores de serviços.

O modelo Rajendran e Balasubramanie faz uma releitura do modelo da W3C apontado

anteriormente e reorganiza os requisitos de QoS, conforme mostra a subseção a seguir.

2.5.4 Modelo de Rajendran e Balasubramanie

O modelo de Rajendran e Balasubramanie (2009) segue os princípios do modelo da W3C e organiza os requisitos de QoS em: acessibilidade, confiabilidade, desempenho, disponibilidade, escalabilidade, integridade, interoperabilidade e precisão.

A acessibilidade de um *Web service* é medida pela probabilidade de uma requisição de um cliente à um *Web service* ser completada. Ainda, pode ser representada como a taxa de sucesso de instanciação de um serviço em determinado período de tempo, ou como a razão entre o número de repostas vindas do servidor e recebidas com sucesso e o número de requisições enviadas pelo cliente, representada pela Equação 1.

$$\text{Acessibilidade} = \frac{\text{Número de confirmações recebidas}}{\text{Número total de requisições enviadas}} \quad (1)$$

A confiabilidade mede o desempenho de um *Web service*, tendo como base uma certa quantidade de tempo e as condições atuais da rede, sem perdas na qualidade do serviço. Além disso, ela determina a porcentagem de vezes que um evento é completado com sucesso.

O desempenho de um *Web service* diz respeito ao quão rápido uma solicitação de *Web service* pode ser processada e finalizada. Este requisito tem como base a taxa de transferência e a latência. Taxa de transferência é a quantidade de requisições concluídas num período de tempo, bem como latência é o tempo que o usuário deve aguardar para obter a resposta de retorno de sua requisição.

Disponibilidade é a probabilidade de um *Web service* estar ativo e funcionando. Como o sistema está sempre ativo ou inativo, podemos utilizar o tempo de inatividade do sistema para calcular a medida de disponibilidade.

Escalabilidade determina o quão expansível um *Web service* pode ser. Da mesma forma, um *Web service* deve ser capaz de controlar alto fluxo de dados, ao mesmo tempo que garante que o desempenho em termos de tempo de espera do cliente não seja afetado.

Integridade garante que quaisquer modificações feitas no *Web service* são executadas por métodos autorizados. Além disso, integridade dos dados é representada pelas habilidades do *Web service* de entrega dos dados e execução precisa de transações. Dessa forma, a integração dos dados pode ser calcula pela Equação 2.

$$\text{Integridade} = \frac{\text{Número de transações com sucesso}}{\text{Número total de transações}} \quad (2)$$

A interoperabilidade diz que um *Web service* pode ser usado por qualquer sistema independentemente de sistema operacional ou arquitetura do sistema e que resultados idênticos e precisos devem ser apresentados em qualquer ambiente. Dessa forma, interoperabilidade pode ser calculada como a razão entre o total de ambientes em que o *Web service* roda e o total de ambientes em que o *Web service* pode ser utilizado, como mostra a Equação 3.

$$\text{Interoperabilidade} = \frac{\text{Número total de ambientes do } \textit{Web service}}{\text{Número total de ambientes possíveis}} \quad (3)$$

A precisão é mensurada pela quantidade de informações corretas entregue pelo *Web service*. Como exemplos temos o número de erros gerados pelo *Web service*, o número de erros fatais e a frequência em que ambos ocorrem.

2.6 REST E ARQUITETURA ORIENTADA A RECURSOS

A estrutura e apresentação do *REST* se assemelha a ROA, uma arquitetura *RESTful*[1] considerada bem estabelecida. A ROA é um modo de transformar um problema em um *Web service RESTful*, rearranjando todas as URIs, HTTPs e XMLs que irão trabalhar como qualquer sistema Web. Esta arquitetura é dividida em recursos, seus respectivos nomes, suas representações e as conexões existentes entre eles e propõe como propriedades o endereçamento, ausência de estado (do inglês, *statelesness*), conectividade (do inglês, *conectedness*) e uma interface uniforme, tendo como objetivo implementar todas estas partes do processo de modo a cumprir todas as propriedades apresentadas (RICHARDSON; RUBY, 2007).

Pode-se dizer que tudo dentro da ROA é considerado *RESTful*, deixando claro que REST não é uma arquitetura, mas sim um conjunto de critérios para design de páginas Web e por ser assim, é muito genérico. Com isso, observa-se que pessoas tendem a apresentar suas próprias arquiteturas quando desenvolvem *Web services*, baseando-se apenas em seus conhecimentos adquiridos sobre REST. Obviamente, o resultado são milhares de *Web services* híbridos REST-RPC, os quais são considerados *RESTful* por seus desenvolvedores. Para melhor entender o que significa REST, abaixo estão as partes que compõem o estilo arquitetural.

[1] serviço baseado em REST

2.6.1 Estrutura

Recurso é algo suficientemente importante para ser representado como um objeto. Geralmente, é algo que pode ser armazenado em um computador e ser representado por uma sequência de bits, como por exemplo um documento, uma linha no banco de dados, entre outros. Recursos podem ser representações de objetos físicos como um carro, ou conceitos abstratos como um sentimento. Mas o que faz um recurso ser um recurso dentro da Web é ele ter pelo menos uma URI.

URI é o nome e endereço de um recurso. Se alguma informação não possui uma URI, ela não é um recurso e não está realmente posicionada dentro da Web, salvos os bits de dados que representam outros recursos. Existiam vários sistemas hyper-texto antes do HTML, e outros protocolos antes do HTTP, mas eles não comunicavam-se entre si. O URI interconectou todos eles dentro da *Web*, simplificando todo o sistema e fez isto de modo simples, pois possuía algo que todos os outros protocolos não tinham que é uma maneira fácil de rotular todo item disponível dentro da Web (WEBBER et al., 2010).

O relacionamento entre estas duas etapas para gerar o estilo REST como é conhecido hoje passa por algumas restrições, onde recursos devem ser únicos, salvos os momentos em que dois recursos diferentes podem estar apontando para um mesmo dado, recursos devem possuir uma URI ou mais, facilitando a referência ao recurso enquanto que dificulta a verificação destas referências e por último, toda URI deve apontar exclusivamente para um recurso, como o nome já diz, *Universal Resource Identifier*, excesso páginas que carregam consigo links para outras páginas Web além do próprio conteúdo da página.

2.6.2 Propriedades

Com o conceito delas agora bem definido pode-se então abordar duas características da arquitetura ROA, endereçamento e ausência de estado definido. Uma aplicação é considerada endereçável se apresenta os aspectos importantes de seu conjunto de dados como recursos e como recursos são expostos por meio de URIs, uma aplicação como esta exibe uma URI para cada informação que consegue compreender. Do ponto de vista do usuário final, endereçamento é o aspecto mais importante de qualquer aplicação ou página Web. Torna fácil ao usuário acessar páginas Web, pois tudo soa muito natural, simples.

Endereçamento é o primeiro das quatro características da ROA, a segunda é *statelessness*. Em termos gerais, quando toda requisição HTTP de um sistema acontece de forma isolada, considerasse que esta arquitetura possui ausência de estado, isto é, quando o cliente faz uma requisição HTTP, ela contém toda a informação necessária para o servidor compreendê-la e processá-la. O servidor nunca confia em informações vindas de requisições passadas. Baseia-se na ideia de que se a informação fosse importante, o cliente teria enviado novamente a requisição (FIELDING, 2000). De forma prática, considera esta ausência de estado definida em termos de endereçamento. Endereçamento diz que toda informação relevante que o servidor possa prover deve ser atribuída a um recurso e receber sua própria URI. *Statelessness* complementa ainda e diz que os possíveis estados do servidor também são recursos e por isso possuem o direito de receber suas próprias URIs. Isso evita problemas comuns em páginas Web onde o botão de voltar do navegador não funciona, pois não consegue percorrer o histórico da navegação (PAUTASSO et al., 2008) .

A compreensão do conceito de representação e sua importância para a arquitetura é essencial para o entendimento de conectividade. Quando dividi-se uma aplicação em recursos, aumenta-se sua área de cobertura e usuários podem construir suas respectivas URI e acessar a aplicação exatamente onde gostariam de estar. Mas estes recursos não são dados, o *Web server* não envia dados, ele envia uma sequencia de *bytes* em um formato de arquivo específico, em uma linguagem específica. Isso é chamado de representação do recurso.

Um recurso é uma fonte de representações e elas são somente um conjunto de dados que simbolizam o estado atual do recurso. E na maioria dos serviços *RESTful*, representações são hipermídia, como documentos que possuem links para outros recursos além do própria informação do documento. Tendo como base o que já foi visto sobre REST até agora e considerando hipermídia como motor que gerencia o estado da aplicação, observa-se que o estado atual de uma sessão HTTP é *rastreado* pelo usuário como um estado da aplicação e criado pelo caminho apresentado pelo usuário para navegar na Web, ao invés de ser armazenada no servidor como um estado do recurso. O servidor passa para o usuário informações sobre os recursos próximos ao recurso atual e como chegar até eles. Esta estrutura de ligações entre as representações dos recursos representa a conectividade de uma arquitetura REST (RICHARDSON; RUBY, 2007).

Uma interface uniforme é a última propriedade da ROA. Ela segue os princípios básicos do HTTP para manuseio de recursos por meio dos quatro métodos de operações: GET, POST, PUT, DELETE. Quando o cliente deseja capturar um recurso, ele envia uma requisição GET para sua URI e quando deseja deletar um recurso existente, envia um DELETE para a URI. No caso da requisição GET, o servidor envia de volta uma representação no corpo entidade da

resposta, enquanto que no DELETE, somente uma mensagem de status é retornada, quando o retorno não é vazio. Além disso, para criar ou alterar um recurso, o usuário envia uma requisição PUT incluindo o corpo entidade na requisição. Esta entidade contém a nova representação do recurso proposta pelo cliente. E por último o mais confuso de todos, o método POST. Este método possui dois propósitos e um deles se encaixa nas restrições do REST. Em uma design *RESTful*, POST é normalmente usado para criar recursos subordinados, isto é, recursos que existem em relação a algum recurso pai. Salienta-se ainda que é uma maneira de criar um novo recurso sem que o cliente saiba exatamente qual é a URI, sendo necessário somente ter conhecimento da URI do recurso pai. O servidor faz o resto pegando a representação do corpo entidade para criar um novo recurso abaixo do recurso pai, variando de um contexto pra outro (RICHARDSON; RUBY, 2007; WEBBER et al., 2010).

2.7 REST ASSURED

O REST Assured é uma biblioteca para testes de serviços REST desenvolvida em Java pela Jayway. Ela introduz a simplicidade de se aplicar testes em *Web services* em linguagens dinâmicas como *Groovy* ou *Ruby*, mas em Java. Não somente é preparada para escalabilidade de casos de teste usando configurações detalhadas, filtros e especificações bem descritas, mas também oferece várias funcionalidades como sintaxe DSL, validação *XPath*[2], reutilização de artefatos provenientes das especificações e facilidade no *upload* de arquivos. Ele simplifica os trabalhos eliminando a necessidade de usar códigos irrelevantes no teste e validação de respostas complexas.

Além disso, possui suporte a requisições e respostas XML e JSON (do inglês, *JavaScript Object Notation*) e a alguns esquemas de autenticação e outros suportes como *OAuth*[3], *Digest*[4] e autenticação de certificados. No quadro 1, um exemplo de sintaxe da versão 2.0 da biblioteca, apresentando o novo método utilizado *given-when-then*, considerado pelos usuários mais natural e familiar para se trabalhar (HALEBY, 2013; KOPS, 2011).

Ainda conforme Haleby (2016a), REST Assured trabalha com JsonPath, ferramenta para análise de documentos escritos em JSON, XmlPath para a análise de documentos em XML, validação de JSON Schema[5], verificando se uma resposta de retorno em JSON está

[2]Sintaxe utilizada para definir as partes de um documento XML
[3]Possibilita o fluxo de autorizações específicas para aplicações Web, *desktop*, celulares, entre outros
[4]Representação de texto em forma de uma sequencia de dados criada usando uma função *hash*
[5]Descreve o formato dos dados em JSON

em conformidade com o JSON Schema, e desenvolvimento de testes de unidades para os controladores quando da utilização de *Spring Mvc*.

Quadro 1 – Exemplo de código em REST Assured

```
given().
      queryParam("lottoId", 5).
when().
      get("/lotto").
then().
      statusCode(200).
      body("lotto.lottoId", equalTo(5)).
      body("lotto.winners.winnerId", hasItems(23, 54));
```

Com o intuito de facilitar a utilização do *framework*, bem como auxiliar na expansão do projeto, Haleby (2016b) elaborou um *GitHub Wiki*[6] do REST Assured. Alguns exemplos de utilização são apresentados a seguir.

Requisições GET com resposta de retorno estruturas em JSON são fáceis de serem verificadas. Por exemplo, a verificação de um *lottoId* ser igual a 5 pode ser resolvida aplicando-se o seguinte código:

Quadro 2 – GET com JSON

```
get("/lotto").
then().
      body("lotto.lottoId", equalTo(5));
```

Não só verifica floats e *doubles*, como também *BigDecimal*, simplesmente alterando a configuração de retorno da requisição. Por outro lado, requisições POST, por exemplo, utilizando resposta de retorno em XML, seguem a mesma estrutura apresentada anteriormente. Para verificar se o primeiro nome é John, por exemplo, um código demonstrativo seria o seguinte:

Quadro 3 – POST com XML

```
given().
      parameters("firstName", "John", "lastName", "Doe").
when().
      post("/greetXML").
then().
      body("greeting.firstName", equalTo("John"));
```

Com o *framework* é possível também realizar buscas em profundidade em documentos XML. Vamos, por exemplo, efetuar a busca do primeiro nó que possui um atributo chamado

[6]Lugar dentro do repositório onde é possível compartilhar conteúdos essenciais do projeto, como a forma de utilização, como foi desenvolvido, entre outros

type igual *groceries*. O exemplo é exemplificado a seguir, com destaque para os asteriscos duplos "**"os quais representam a busca em profundidade:

Quadro 4 – GET com busca em profundidade e XML

```
when().
    get("/shopping").
then().
    body("**.find { it.@type == 'groceries'}", hasItems("Chocolate", "Cofee"));
```

Verificar os dados do corpo de resposta é outra funcionalidade do *framework*. Com ele, pode-se aferir o *status code*, *status line*, *cookies*, cabeçalhos, tipos de conteúdo e corpo de resposta. Além disso, pode ser efetuado o mapeamento do corpo de resposta para um objeto Java. Alguns exemplos são apresentados a seguir:

Quadro 5 – Verificando os dados do corpo de resposta

```
get("/x").
then().
    assertThat().cookie("cookieName", "cookieValue");
```

```
get("/x").
then().
    assertThat().statusCode(200);
```

```
get("/x").
then().
    assertThat().body(equalTo("something"));
```

```
get("/x").
then().
    assertThat().contentType(ContentType.JSON);
```

Por fim, outras funcionalidades da ferramenta são: medir tempo de resposta, autenticação básica e OAuth2, dados de formulários *multi-part*, mapeamento de objetos como serialização e desserialização, analisadores de dados, reuso de especificações, filtros, logging, suporte de sessão, *Secure Socket Layer* (SSL), configuração de *proxies*, entre outros.

3 MATERIAL E MÉTODOS

O *framework* para validação de *Web services RESTful* REST Assured é um Java DSL aberto que evita a utilização de códigos grandes e desnecessários no momento de testar respostas complexas de APIs (do inglês, *Application Programming Interface*) e ainda provém suporte a ambas as tecnologias XML e JSON. Com REST Assured pode-se:

- executar trocas de fontes de dados externas por dados predefinidos, técnica denominada *stubbing*;
- verificar requisições de dados externos;
- trabalhar com esquemas de autenticação utilizando OAuth;
- fácil *parsing* de retorno do corpo de resposta para JSON e XML;
- gerar *logs* detalhados de requisições e respostas com filtros predefinidos;
- simular comportamentos de serviços externos de várias formas usando uma interface Web;
- executar testes de unidade em controladores *Spring MVC*.

Quanto a maneira de escrever os testes, REST Assured utiliza a tradicional estrutura *Given/When/Then* oriunda do BDD, considerada uma versão refinada e sintetizada das práticas de TDD e ATDD. A estrutura descrita em tópicos:

- *Given*: especificação dos parâmetros para as chamadas da API;
- *When*: momento da chamada da API;
- *Then*: verificação se a resposta é o retorno desejado;

O *framework* roda em um processo *standalone* e pode ser configurado durante tempo de execução para executar e responder a qualquer requisição com conteúdos aleatórios e arbitrários, status, cabeçalhos, entre outros. Pode também ser combinado com outros *frameworks* de teste como o TestNG, obtendo-se um *framework* de testes de serviços REST muito mais encorpado e poderoso (HALEBY, 2013).

Abaixo um trecho de código exemplificando o uso da biblioteca em Java.

Quadro 6 – Trecho de código em Java

```
@Test
public void testCreateUser() {
    final String email = "thiago@trevisan.com";
    final String firstName = "Thiago";
    final String lastName = "Trevisan";

    given().
        parameters(
            "email", email,
            "firstName", firstName,
            "lastName", lastName).
    when().
        get("/user/create").
    then().
        body("email", equalTo(email)).
        body("firstName", equalTo(firstName)).
        body("lastName", equalTo(lastName));
```

3.1 FERRAMENTAS

Esta seção descreve as ferramentas que auxiliaram no desenvolvimento deste trabalho. São elas: Java e Maven. Em Java, apresenta-se um pouco da história e funcionamento da linguagem, bem como discorre-se sobre o paradigma orientado a objeto, enquanto que em Maven, apresenta-se uma breve definição do software.

3.1.1 Java

Java surgiu na década de 90 com o intuito de solucionar alguns problemas que apareciam com frequência nas linguagens já existentes no mercado, como ponteiros, gerenciamento de memória, falta de bibliotecas, portabilidade, entre outros.

A linguagem foi criada pela antiga Sun Microsystems com o objetivo de que essa linguagem fosse usada em pequenos dispositivos, como televisores, aspiradores, liquidificadores, entre outros. No entanto, ela teve seu lançamento focado em rodar pequenas aplicações (*applets*) em navegadores, tendo os clientes Web como público-alvo. E por fim,

ambos os objetivos foram perdendo a preferência ao longo dos anos e o foco da companhia passou a ser no desenvolvimento de aplicações do lado do servidor, a marca Java ficou fortalecida após a compra da Sun pela Oracle em 2009 e em 2014, surge a nova e mais atual versão do Java, o Java 8 (CAELUM, 2014).

Uma das vantagens do Java e que o tornou popular é a portabilidade oferecida pela linguagem, não existente em outras linguagens como C e Pascal. Nestas linguagens, a aplicação comunica-se diretamente com o sistema operacional (SO), portanto o código fonte é compilado para um código de máquina específico deste sistema ou plataforma. Entretanto, Java trabalha com o conceito de máquina virtual. Neste modelo, não existe comunicação direta da aplicação com o SO, pois todo código fonte é compilado por um compilador Java, como o *javac*, para gerar um *bytecode* que será interpretado por uma camada existente entre a aplicação e o SO, denominada *Java Virtual Machine* (JVM). Esta camada é responsável por interpretar o que a aplicação deseja fazer para as respectivas chamadas do SO onde ela está rodando e por aumentar a performance do sistema por meio da compilação dinâmica da JVM que identifica quando um código não possui o desempenho adequado e otimiza aquele trecho de código (CAELUM, 2014; PERRY, 2015).

Como qualquer outra linguagem de programação, a linguagem Java possui sua própria estrutura, regras de sintaxe, e paradigma de programação. Por ser derivada da linguagem C, possui regras de sintaxe muito similares a ela, como por exemplo, os blocos são modulados em métodos e delimitados por chaves, e as variáveis são declaradas antes de serem utilizadas, veja Código 1. Ainda, estruturalmente Java é organizado em pacotes, os quais representam o mecanismo *namespace* da linguagem. Dentro destes, existem as classes com seus atributos, métodos, constantes, entre outros. Por último, o paradigma de programação do Java é baseado no conceito de programação orientada a objetos (OOP, sigla em inglês) (PERRY, 2015).

```
1  public class Pessoa {
2
3      private long id;
4      private String name;
5      private String rg;
6
7      public long getId() {
8          return id;
9      }
10
11     public void setId(long id) {
12         this.id = id;
13     }
14
15     public String getName() {
16         return name;
17     }
18
```

```
19      public void setName(String name) {
20          this.name = name;
21      }
22
23      public String getRg() {
24          return rg;
25      }
26
27      public void setRg(String rg) {
28          this.rg = rg;
29      }
30
31  }
```

Código 1 – Java

3.1.1.1 Paradigma

Segundo Perry (2015), a orientação a objetos combina os dados e as instruções do programa em objetos. Um objeto é tido como uma entidade autônoma que possui atributos e comportamentos, e utilizam de mensagens (chamadas de métodos, na terminologia Java) para se comunicar com outros objetos. Tem por objetivo modelar objetos do mundo real, pois considera que todos os objetos do mundo real compartilham de duas características, estado e comportamento. Cachorros, por exemplo, possuem estados: nome, cor, raça, fome e possuem comportamentos: latir, buscar, abanar rabo, entre outros. Comumente armazena-se os estados do objeto em campos (ou variáveis) e expõe seus comportamentos por meio de métodos. Este métodos operam sobre o estado interno do objeto e servem como principal comunicação entre objetos (ORACLE DOCS, 2015a).

Os três princípios deste paradigma são o encapsulamento, herança e polimorfismo. Encapsulamento dos dados ocorre quando os estados do objeto são privados, isto é, somente podem ser manipulados pelo próprio objeto, e todas as interações com o objeto são efetuadas por meio dos métodos do objeto. Quando temos que dois ou mais objetos compartilham algumas de suas características, aplicamos a herança, onde tem-se uma superclasse, ou classe pai, que possui os estados e comportamentos em comum entre os objetos, e subclasses, ou classes filho, que possuem seus estados e comportamentos específicos, podendo sobrescrever algum estado e/ou comportamento da superclasse na subclasse se necessário (PERRY, 2015; ORACLE DOCS, 2015a). Polimorfismo, considerado um conceito mais complexo que encapsulamento e herança, tem por essência que objetos pertencentes a mesma estrutura de hierarquia, podem manifestar comportamentos diferentes quando envia-se uma mesma mensagem. Na terminologia Java, temos que subclasses da classe podem definir seus próprios comportamentos e ainda compartilhar algumas das funcionalidades da classe pai (PERRY, 2015; ORACLE DOCS, 2015b).

3.1.2 Maven

Apache Maven é uma software de gerenciamento de projetos que possui sua estrutura baseada no conceito de POM. POM é um arquivo XML que contém a informação sobre o projeto, bem como os detalhes da configuração utilizado pelo Maven para executar a construção do projeto (ENES, 2007). Com o Maven é possível gerenciar a construção do projeto, criar relatórios e documentação por meio de uma estrutura centralizada. Surgiu como uma solução padronizada para construir e gerenciar projetos baseados em Java, bem como fornecer uma forma facilitada de publicar informações do projeto e compartilhar JARs entre diversos projetos. A ferramenta tem por principal objetivo permitir ao desenvolvedor compreender a camada completa do estado de desenvolvimento no menor período de tempo possível (THE APACHE SOFTWARE FOUNDATION, 2016). E para alcançar estes objetivos, algumas áreas recebem especial atenção:

- Tornar simples o processo de construção, eliminando a necessidade de conhecer todas as minúcias do projeto;
- Oferecer um sistema de construção uniforme, permitindo a construção do projeto a partir do POM e de um conjunto de *plugins*;
- Prover informações de qualidade sobre o projeto, geradas por meio do próprio POM e gerados a partir dos códigos fonte, como por exemplo, arquivos de registro *log* das mudanças, fontes com referências cruzadas, listas de email e de dependências, entre outras;
- Fornecer um guia para as boas práticas de desenvolvimento, por meio da especificação, execução e geração de relatórios de testes de unidade, bem como auxiliar no *workflow* do projeto, gerenciando os *releases* e mantendo controle das *issues* geradas;
- Permitir a migração de forma facilitada para novos recursos, fornecendo uma forma fácil de atualizar as instalações de pacotes de terceiros e do próprio Maven.

3.2 MÉTODOS DE APLICAÇÃO

Os testes aplicados sobre o sistema de uma empresa real e apresentados neste trabalho tem por objetivo prover suporte a QoS em *Web services*.

Uma reunião com os membros da equipe da empresa foi organizada onde todos os modelos de QoS foram apresentados com o intuito de situar os membros quanto so suporte a QoS. Foi especificado durante esta reunião que três dos requisitos entre todos os modelos seriam selecionados para a aplicação neste trabalho.

A ideia de permitir a mescla de modelos de qualidade de produto surgiu com o intuito de possibilitar aos membros maior flexibilidade de escolha, podendo assim selecionar o requisito com a definição que melhor se encaixaria nas necessidades da empresa atualmente. Com outras palavras, uma forma de capturar o melhor para a empresa de cada requisito abordado pelos modelos.

Ao fim da reunião, a equipe optou focar nos requisitos de *desempenho*, *integridade* e *segurança*, ou seja, consideraram estes os principais pontos a serem validados com o intuito de detectar falhas, se existentes. Uma breve descrição de cada uma delas é apresentada a seguir, destacando-se que de forma geral, optou-se por trabalhar com os requisitos abordados pela W3C que possuem foco em qualidade de *Web services*.

Em desempenho, os testes seguem a definição apresentada pela W3C e pelos engenheiros Rajendran e Balasubramanie, onde desempenho representa o quão rápido uma requisição de serviço pode ser completada, mantendo o foco no tempo de resposta e latência do serviço. Além disso, adiciona-se neste requisito os tópicos de comportamento em relação ao tempo e utilização dos recursos apontados pela ISO 25010.

Ainda, em integridade os testes também seguem o modelo da W3C em que o sistema deve prevenir e impedir o acesso não autorizado ou modificações dos dados, com ressalvas para o fato de que na ISO 25010, integridade faz parte do requisito *segurança*, mas neste trabalho, ele é tratado com um requisito de QoS.

Por fim, segurança é abordada neste trabalho seguindo as especificações da ISO 25010, que descreve segurança como grau de proteção do sistema pelo qual pessoas e outros produtos tenham acesso aos dados com base em seus tipos e níveis de autorização apropriados. Salienta-se ainda o foco em suas características específicas de confidencialidade, não-repudiação, responsabilidade e autenticidade.

Deste modo, em reunião com a equipe da empresa, levanta-se possíveis situações dentro das funções abordadas neste trabalho com o intuito de estabelecer hipóteses para as possíveis situações que podem ocorrer nos serviços.

Posteriormente, com base nas hipóteses levantadas, desenvolve-se os casos de teste que abordam somente partes das funções do serviço, formando assim os testes de unidade. Além disso, técnicas de *script* modular existentes no *framework* são empregadas para organização dos casos de teste em módulos que representam as funções do serviço.

3.3 AMBIENTE DE APLICAÇÃO

Os testes são executados utilizando um notebook com processador i7 de quarta geração e 8GB de memória, bem como foi utilizado um servidor local para a realização do trabalho. O acesso ao servidor ocorre por meio de conexão de internet de alta velocidade protegida por *proxy*.

3.4 APLICAÇÃO

O sistema sobre teste tem como meta oferecer um serviço que facilite a localização de eventos na região onde o cliente estiver e tem como principais objetivos, a divulgação de eventos e a venda de ingressos para eventos de um modo fácil e rápido.

A API e o ambiente *Web* está sendo desenvolvido em Node.js por meio do *framework* Express enquanto que o desenvolvimento *mobile* está sendo feito por meio do *framework* Apache Cordova.

A equipe da empresa que auxiliou no desenvolvimento deste trabalho, isto é, os *stakeholders*, é composta por 3 membros: um responsável pela API, outro pela interface *Web* e o último pelas aplicações *mobile*.

Destaca-se ainda que o sistema ainda está em processo de desenvolvimento, portanto, nem todos os serviços apresentados no modelo de negócios da empresa foram finalizados.

3.5 HIPÓTESES

As hipóteses (*stories*) levantadas para este trabalho foram selecionadas juntamente com a empresa com o intuito de aplicar o modelo apresentado pelo método BDD, gerando as *stories* e os seus possíveis *scenarios*, e propiciar um resultado satisfatório para ambas as partes. Além disso, as descrições textuais de cada uma das hipóteses são apresentadas como proposto

pela equipe, mirando uma alta fidelidade. Além disso, elas estão organizadas de acordo com o requisito ao qual pertencem e são descritas a seguir.

O requisito de desempenho trabalha com valores qualitativos, não mensuráveis. Deste modo, estipula-se para este trabalho que os valores de tempo de resposta são subjetivos e cabe ao leitor interpretar os valores, se eles estão de acordo ou não com o esperado. Comentários particulares são apresentados mas não devem ser utilizados como base para análise. As hipóteses são:

Quadro 7 – Hipóteses de desempenho

O acesso aos eventos pelo cliente deve ser rápido;
O tempo de espera para visualização dos eventos deve ser o mínimo possível;
O tempo de login deve ser imperceptível;
O usuário não deve esperar muito para ser encaminhado para a finalização da compra por sistema terceirizado.

Integridade abordada aqui foca na autorização dos usuários para acesso à determinadas funcionalidades do sistema, bem como a garantia da integridade das transições. As hipóteses são:

Quadro 8 – Hipóteses de integridade

Somente usuário autorizado como administrador pode acessar a base de dados do sistema;
Somente usuário autorizado como administrador pode efetuar alterações de dados na base de dados do sistema;
Usuário deve estar logado para efetuar a compra do ingresso;
O encaminhamento para a finalização da compra por sistema terceirizado deve gerar a URL esperada.

O requisito de segurança trabalha com a confidencialidade dos dados, na responsabilidade das ações tomadas e na autenticidade do serviço. As hipóteses são:

Quadro 9 – Hipóteses de segurança

Usuários cadastrados podem alterar somente os próprios dados;
Dados dos eventos podem ser alterados somente pelo promotor do evento;
Cada usuário deve acessar somente seus próprios ingressos;
Cada usuário deve acessar somente suas próprias transações;
A compra do ingresso deve identificar se o usuário que efetuou a transação é dono da conta.

3.6 CASOS DE TESTE

A descrição dos testes é apresentado nesta seção com o intuito de expor os métodos utilizados no desenvolvimentos dos testes que representam as hipóteses estabelecidas anteriormente. Destaca-se ainda que não foram desempenhados casos de testes para todos itens levantados, bem como a URI base para todos os casos de testes apresentados a seguir é o endereço `http://localhost:1337/api`.

Por fim, os valores utilizados como referência foram apresentados pelos próprios membros da empresa, com base em testes previamente realizados pela equipe em servidores locais. Além disso, são valores que a própria equipe considera ser um bom limiar de referência para o sistema, considerando ainda um servidor local.

Realça-se ainda o fato de que nenhum valor base de referência foi encontrado nas normativas e modelos de QoS apresentados para os tópicos abordados.

3.6.1 Desempenho

Para o requisito de desempenho foram preparados dois *Mock Objects* para base de dados dos testes. Primeiro, três conjuntos de objetos foram criados, com cem (100), quinhentos (500) e mil (1000) eventos. Segundo, um único objeto foi gerado representando um usuário cadastrado no sistema.

Alguns parâmetros foram levantados juntamente à equipe da empresa para mensurar os dados qualitativos mencionados. Para as consultas aos eventos foi estabelecido os tempos de resposta de 90, 95 e 100 milissegundos (ms), respectivamente, para o tempo de resposta do login, 500 ms, e para a confirmação de pagamento, 3000 ms. Além disso, para apresentar dados mais precisos, cada teste foi executado dez vezes, bem como valores máximos e mínimos foram retirados da amostra. Com a amostra restante, calculou-se a média aritmética para gerar o valor final.

Com o objetivo de verificar o tempo de acesso aos eventos pelo cliente foram preparados três casos de teste: teste sem autenticação, autenticado e verificação se atingiu os parâmetros. O primeiro teste teve o intuito de verificar o tempo de resposta da busca de eventos executados por meio de requisição GET sem autenticação de usuário. Começou-se o processo

por especificar a URI base dos casos de testes. Pelo fato de não necessitar de autenticação, nenhum parâmetro de entrada foi atribuído a requisição. Logo após, foram especificados a rota da requisição e o parâmetro de cálculo do tempo de resposta exercido pela requisição, especificando que o valor seria apresentado em milissegundos. Os resultados médios são 93, 93,875 e 94 ms respectivamente seguindo os 100, 500 e 1000 eventos e os desvios padrão são 1,690, 5,693 e 0,756. Os valores são apresentados na Tabela 1.

O teste seguinte teve a mesma lógica do primeiro, mas foi especificado dois parâmetros de entrada pelo fato de representarem o usuário autenticado pelo sistema. Primeiro, apresentou-se no cabeçalho o *Content-Type*, informando ao servidor que tipo de mídia ele iria tratar e como ele irá tratá-lo, representado neste teste pelo formato JSON, isto é, *application/json*, e por último o *Authorization*, campo composto de credenciais contendo a informação de autenticação do usuário para o domínio do recurso a ser requisitado com o intuito de autenticar a entidade juntamente ao servidor, representado neste teste pelo *token* de acesso gerado pela autenticação do usuário (FIELDING; RESCHKE, 2014b; FIELDING; RESCHKE, 2014a). Os resultados médios são 88,125, 92,375 e 102,5 respectivamente e os desvios padrão são 4,257, 9,486 e 7,728. Os valores são mostrados na Tabela 1.

O terceiro teste teve por finalidade verificar se cada um dos testes executados acima cumpriram os objetivos de desempenho preestabelecidos, levando a estabelecer uma medida de desempenho do sistema nesta *story* representada pela razão entre o número de sucessos e o número total de testes efetuados, como mostra a Equação 4. Os resultados que mais se destacaram foram a busca de 1000 eventos sem autenticação que alcançou 100% das metas e a de 100 eventos sem autenticação com 0%. Os testes são apresentados na tabela 2.

$$\text{Desempenho} = \frac{\text{Tempo de acesso dentro dos parâmetros}}{\text{Número total de testes}} \tag{4}$$

A visualização dos eventos é, segundo conversa com a equipe da empresa, desenvolvida em sua totalidade no *front-end* do sistema. Como não havia forma de desenvolver casos de teste para serem aplicados em serviços REST, um dos objetivos do trabalho, a *story* não foi avaliada.

Para efetuar a verificação do tempo de login do *Web service* foram organizados dois casos de testes: teste para verificar o tempo de resposta e verificação de cumprimento do parâmetro preestabelecido. O teste de cálculo do tempo de resposta do serviço para efetuar o login do usuário foi preparado utilizando uma requisição POST, passando como parâmetros, o e-mail e senha do usuário. Primeiramente, a URI base dos casos de testes foi atribuída ao corpo do teste. Posteriormente, um arquivo no formato JSON foi preparado com o intuito de ser enviado juntamente com a requisição do serviço, com destaque para o fato de que neste

trabalho, o arquivo foi representado por uma *string* de dados. Em seguida, a requisição é estruturada pelos parâmetros de entrada, local onde o arquivo JSON foi inserido, seguidos da rota de requisição do serviço e do parâmetro de cálculo do tempo de resposta exercido pela requisição, especificando que o valor foi apresentado em milissegundos. O resultado médio obtido pelo login foi de 509,75 ms e o desvio padrão foi de 3,536. OS valores são descritos na Tabela 3.

O segundo teste teve por finalidade avaliar se cada um dos testes de verificação do tempo de login cumpriram o objetivo de desempenho preestabelecido, levando a estabelecer uma medida de desempenho do sistema nesta *story* representada pela razão entre o número de testes dentro dos parâmetros e o número total de tentativas, como mostra a Equação 4. Os resultados mostram que nenhum dos testes alcançou a meta de 500 ms previamente estabelecidas. Os valores dos testes são apresentados na Tabela 4.

Com o intuito de verificar o tempo de espera do usuário para ser encaminhado para a etapa final da compra foram desenvolvidos dois casos de teste: mensurar o tempo de resposta da validação da compra e verificar se cumpriu o parâmetro estabelecido. Para calcular o tempo de resposta do encaminhamento para sistema terceirizado foi criado uma requisição POST tendo um usuário autenticado no sistema. O processo começou pela adição da URI base para os casos de teste. Logo após, são apresentados os dois cabeçalhos que orientam e guiam a comunicação entre o servidor e o requisitor do serviço e o corpo da requisição, *Content-Type*, *Authorization* e o arquivo JSON, respectivamente. Neste arquivo, são incluídos, as informações do itens comprados e os dados do usuário que está efetuando a compra. Por fim, a rota da requisição e do parâmetro para medir o tempo de resposta gerado pela requisição. O resultado médio foi de 2280,625 ms com um desvio padrão de 537,806. Os valores são exibidos na Tabela 3. Destaca-se que neste caso de teste, somente foi contado o tempo até o retorno de resposta do serviço. Tempos de redirecionamento, tanto de páginas Web quanto aplicativos não foram incluídos nos cálculos.

O segundo teste teve por finalidade avaliar se cada um dos testes efetuados acima cumpriram o objetivo de desempenho preestabelecido, levando a estabelecer uma medida de desempenho do sistema neste *scenario* representada pela razão entre o número de testes dentro dos parâmetros e o número total de tentativas, como mostra a Equação 4. Os resultados mostram que foram obtidos diversos valores abaixo dos 2000 ms, mas também foram encontrados valores superiores a meta de 3000 ms previamente estabelecida. Os testes são apresentados na Tabela 4.

3.6.2 Integridade

Para o requisito de integridade foram preparados dois *Mock Objects* para base de dados dos testes, representando um usuário cadastrado no sistema e um evento com ingressos variados. Além disso, para a apresentação dos dados de encaminhamento para a finalização da compra de forma precisa, o teste foi dividido em três conjuntos, de dez (10), cinquenta (50), e cem (100) confirmações de compra executadas de forma sequencial. Com base nos dados da amostra, calculou-se a integridade do serviço seguindo o modelo de Rajendran e Balasubramanie representado pela Equação 2.

Por fim, foram utilizados *HTTP Response Status Codes*, isto é, códigos compostos por três dígitos que representam o resultado da tentativa de compreender e satisfazer uma requisição (FIELDING; RESCHKE, 2014b). Estes códigos são organizados em cinco categorias que descrevem a classe da resposta de retorno, representados pelo primeiro digito do código:

- 1xx: informativo;
- 2xx: bem sucedido;
- 3xx: redirecionamento;
- 4xx: erro do lado do cliente;
- 5xx: erro do lado do servidor.

Conforme conversas com os membros da empresa, foi declarado que o acesso à base de dados ainda não possui uma interface gráfica para gerenciamento, sendo necessário o acesso remoto ao servidor por protocolo de rede *Secure Shell* (SSH), que autentica as credenciais do usuário com o intuito de fornecer um acesso seguro à sistemas remotos. Portanto, não existem ainda serviços REST implementados que permitam acesso à esses recursos. Deste modo, as duas primeiras *stories* deste requisito não foram testadas neste trabalho.

Com o objetivo de verificar se o usuário possui autorização para a efetiva compra do ingresso foram preparados três casos de teste: teste com *token* de acesso válido, com *token* de acesso inválido e sem *token* de acesso. Assim, o primeiro teste tem como objetivo testar a compra do ingresso por meio de uma requisição POST utilizando de um usuário autenticado no sistema, tendo assim um *token* de acesso válido. O processo começou especificando a URI base para os casos de teste, seguidos de ambos os cabeçalhos *Content-Type*, *Authorization*, detalhando que a meio de comunicação seria via JSON, e descrevendo o *token* de acesso, respectivamente. Um arquivo JSON, vem logo em seguida, identificando as informações dos itens da compra e os dados do usuário requisitor do serviço. Posteriormente é adicionado a rota de requisição do serviço, bem como especificado que deve-se obter como código de

retorno o *status code* 200. Este código representa que a requisição foi completada com sucesso (FIELDING; RESCHKE, 2014b).

O segundo teste foi desenvolvido com o mesmo raciocínio do teste anterior, com exceção para o cabeçalho *Authorization* que apresenta um valor de *token* de acesso inválido e o código de retorno, que deve retornar o *status code* 401, indicando que a requisição não foi aplicada pela falta de credenciais para autenticação válidas para o recurso (FIELDING; RESCHKE, 2014a).

Para efetuar o teste de compra do ingresso sem um usuário autenticado no sistema, isto é, sem um *token* de acesso, foi preparado um teste de requisição POST. A URI base para os casos de teste foi adicionada ao teste, bem como o cabeçalho *Content-Type*: *application/json* e o corpo da requisição com os dados referentes à compra, em formato JSON. A rota de requisição do serviço é selecionada a seguir para especificar o momento da chamada. Além disso, especificou-se que o código de retorno deveria apresentar o mesmo *status code* 401 do teste anterior.

Com o intuito de verificar se o sistema gerava a URL esperada para a finalização da compra foi preparado três casos de teste. Todos possuem a mesma estrutura, mas variam na quantidade de confirmações de compra gerados, conforme mencionado anteriormente. Para a construção dos testes foi configurado a URI base para os casos de teste e construído o laço de repetição para geração das confirmações de compra. Dentro do laço é montado a *string* de dados representando um arquivo JSON, bem como uma das funcionalidades do REST Assured para a extração de informações da resposta de retorno do serviço. Além disso, a resposta é formatada utilizando o *JsonPath*, recurso do próprio *framework* para *parsing* do retorno do corpo de resposta para formato JSON, e com este mesmo recurso é extraído a URL de redirecionamento, salvando-a em uma lista para posterior verificação.

A estrutura para extração de informações foi composta por três parâmetros de entrada, sendo dois cabeçalhos *Content-Type* e *Authorization*, e o corpo da requisição onde é inserido a *string*. Logo após configurou-se a rota de requisição e especificou-se que a resposta do retorno seria extraída para posterior análise. Os resultados foram bem sucedidos devido a fato de todos os testes terem alcançado a meta preestabelecida. OS valores são apresentados na Tabela 5.

3.6.3 Segurança

Para o requisito de segurança foi criado um *Mock Object* além dos já mencionados

representando mais um usuário cadastrado no sistema. Nenhum outro tipo de avaliação foi aplicado neste requisito além da execução dos testes e tomada de notas dos resultados. Além disso, os *status codes* apresentados no requisito anterior também serão aplicados aqui.

Com o objetivo de verificar se um usuário poderia alterar somente seus próprios dados foi desenvolvido um caso de teste que utiliza um usuário *user 1* cadastrado no sistema para alterar algum dado pessoal da conta de um usuário *user 2* cadastrado no sistema. Diante disso, o teste foi construído em duas etapas. Primeiro, uma busca por um usuário cadastrado no sistema foi executado por meio de uma requisição GET, onde dados de entrada são especificados com as credenciais de autenticidade do *user 1*. Ainda, a rota de requisição foi adicionada ao teste, bem como descrito que o código de retorno deveria ser *status code* 200. Por fim, foi efetuado a extração de informações da resposta de retorno do serviço para ser utilizado na próxima etapa. Segundo, uma requisição PUT foi desenvolvida para executar a efetiva alteração dos dados. Nesta requisição, os dados de entrada são os mesmos dados do *user 1* utilizados anteriormente seguido do corpo da requisição com um dado pessoal do *user 2* com seu valor alterado. Logo após, a rota de requisição foi especificada, bem como o código de retorno *status code* 401 foi selecionado com o intuito de demonstrar que esta ação não deveria ocorrer.

Para verificar se o promotor do evento seria o único que poderia alterar os dados dos eventos foi organizado somente um caso de teste que tem por finalidade verificar se seria possível que um usuário *user 1* pudesse alterar algum dado de um evento organizado pelo *user 2*. Para tanto, foi preparado uma requisição PUT contendo as credenciais com a informação de autorização do *user 1* e o corpo de requisição como dados de entrada. O corpo de requisição foi representado por um arquivo formato JSON contendo algum dado do evento à ser alterado. A rota de requisição foi estabelecida a seguir, especificando o evento alvo do teste. Por fim, o código de retorno *status code* 401 foi especificado com o mesmo princípio apontado pelo teste anterior.

O acesso aos ingressos do usuário foi avaliado em dois casos de teste: um usuário *user 1* tenta acessar os dados dos ingressos de um usuário *user 2* e um usuário não autenticado tenta acessar os dados dos ingressos de um usuário autenticado. O primeiro teste foi composto por uma requisição GET que recebeu como parâmetros de entrada o tipo de mídia utilizado e as credenciais que autenticam o *user 1* no sistema. A rota de requisição vem logo em seguida com os dados que representam os ingressos referentes ao *user 2*. O código de retorno *status code* 401 foi especificado pelo fato do usuário não possuir teoricamente a permissão para efetuar a ação.

O segundo teste focou em um usuário não autenticado pelo sistema e teve a mesma lógica do primeiro, mas nenhum parâmetro de entrada foi atribuído a requisição. O mesmo

código de retorno *status code* 401 foi especificado para o teste, bem como a mesma rota de requisição.

Para verificar se o acesso as transações do usuário podem ser feitas unicamente por ele, foi preparado dois casos de teste que seguem os mesmos princípios da *story* anterior: um usuário *user 1* tenta acessar os dados das transações de um usuário *user 2* e um usuário não autenticado tenta acessar os dados das transações de um usuário autenticado. Primeiro, uma requisição GET foi construída utilizando os cabeçalhos *Content-Type* e *Authorization* como parâmetros de entrada, apresentando os valores *application/json* e as credenciais de autenticação do *user 1*, respectivamente. Logo em seguida, a rota de requisição é especificada com o intuito de determinar o momento da chamada da API. Por fim, estabeleceu-se que o código de retorno *status code* 401 deveria ser exibido pelo fato de que essa ação não deveria acontecer.

O segundo teste foi reproduzido considerando a tentativa de um usuário não cadastrado no sistema de acessar as transações de algum usuário cadastrado no sistema. Ele segue o mesmo raciocínio do teste anterior, mas abdica dos parâmetros de entrada, pois não possui credenciais válidas com informação de autenticação do sistema. Além disso, o código de retorno *status code* 401 também foi especificado para este teste.

Conforme reunião com os membros da equipe, foi mencionado que a única identificação existente da compra do ingresso é provida pela conta que efetuou a compra do ingresso. Isto ocorre pelo fato de que a finalização da compra por enquanto é efetuada em um sistema terceirizado e dados do cartão do cliente não são armazenados no banco de dados da empresa. Além disso, apontou-se que futuras versões do sistema permitirão a compra direta pelo sistema da empresa, aplicando assim os devidos testes. Em vista disso, esta *story* não foi avaliada neste trabalho.

4 RESULTADOS E DISCUSSÕES

Os resultados e discussões apresentados nesta seção tem o intuito de expor os resultados obtidos após a aplicação dos casos de testes mencionados anteriormente, bem como expor uma avaliação estatística dos valores obtidos e a própria avaliação da equipe da empresa quanto aos resultados.

4.1 DESEMPENHO

Os testes para verificar o tempo de acesso aos eventos foram executados conforme estabelecido na *story*. Além disso, ambos os testes obtiveram resultados satisfatórios, confirmando a hipótese de que os eventos podem ser acessados por usuários autenticados e não autenticados. Os resultados apresentados na Tabela 1 mostram o tempo médio de acesso aos eventos em milissegundos, organizando-os em grupos divididos pela quantidade de eventos incluídos na requisição do serviço, bem como o desvio padrão e a mediana. Apresenta ainda os resultados para as requisições de usuários autenticados (A) e não autenticados (NA).

Eventos	A / NA	Média	Desvio padrão	Mediana
100	A	88,125	4,257	89
	NA	93	1,690	93
500	A	92,375	9,486	91
	NA	93,875	5,693	94
1000	A	102,5	7,728	102,5
	NA	94	0,756	94

Tabela 1 – Tempo de acesso aos eventos.

Os resultados mostram que a média obtida para 100 eventos de 88,125 foi estatisticamente satisfatória para A enquanto que a média de 93 foi negativa para usuários NA. O desvio padrão ficou em 4,257 para A e 1,690 para NA e a mediana em 89 para A e 93 para

NA. Para tanto, mostra que apesar da média de NA ser positiva, existem valores que ficaram acima da meta preestabelecida de 90 ms. Para 500 eventos, ambas as médias de 92,375 para A e 93,875 para NA alcançaram a meta preestabelecida de 95 ms. O desvio padrão foi de 9,486 e 5,693, respectivamente enquanto que a mediana foi de 91 e 94. Com estes valores percebe-se que ambas as médias e medianas permaneceram abaixo dos valores previamente estabelecidos, no entanto, os desvios padrão mostram que existem valores que ficaram relativamente acima do limite, avançando ainda o limite estipulado para a busca de 1000 eventos. Por fim, os resultados para 1000 eventos mostram que a média para A estabeleceu-se acima dos 100 ms previamente estabelecidos com 102,5 enquanto que para NA foi de 94. Os desvios padrão foram de 7,728 e 0,756 para A e NA, respectivamente, e a mediana 102,5 e 94. Por tanto, mostra que para usuários A a busca apresentou resultados ligeiramente negativos, com somente alguns valores permanecendo abaixo da meta devido ao valor de desvio apresentado e que para usuários NA, os valores foram positivos com pequenos desvios que não produziram valores acima da meta previamente estabelecida.

Do ponto de vista da equipe da empresa, os resultados mostram que as médias de tempo de acesso aos eventos estão de acordo com os valores preestabelecidos, mesmo que de forma geral, apresentam estatisticamente valores negativos. Notou-se ainda que a média do tempo de acesso de A aumentou levemente comparado a NA conforme a quantidade de eventos da requisição aumentavam. Supõe-se que este aumento ocorreu pelo fato da API verificar a autenticidade do usuário em cada evento adicionado ao corpo de resposta do retorno.

O teste de desempenho de acesso aos eventos foi desenvolvido pare estabelecer a qualidade do serviço com base nos tempos de acesso preestabelecidos juntamente com a equipe da empresa. Os valores alcançaram ou obtiveram resultados próximos aos especificados na maioria dos testes, destacando-se para os testes de mil eventos com usuário não autenticado que alcançaram a meta em todas as execuções. Os resultados apresentados na Tabela 2 mostram o tempo de resposta do serviço para o acesso aos eventos em milissegundos, organizando-os pela quantidade de eventos incluídas na requisição bem como pela autenticidade dos usuários no momento da requisição do serviço. Além disso, as porcentagens apresentadas na parte de baixo da tabela representam a porcentagem dos testes que alcançaram os valores preestabelecidos em conversa com os *stakeholders*.

Os resultados mostram que 75% das buscas de 100 eventos por usuários A alcançaram a meta dos 90 ms, enquanto que 0% das medidas por usuários NA alcançaram o mesmo valor, apresentando, portanto, resultados extremamente negativos do ponto de vista estatístico. Na busca de 500 eventos, por outro lado, consultas com usuários A obtiveram 75% dos valores positivos, isto é, abaixo dos 95 ms, e 50% com usuários NA. Para tanto, mostra que de forma

100		500		1000	
A	**NA**	**A**	**NA**	**A**	**NA**
93	91	110	95	98	93
90	92	80	100	97	95
92	91	101	102	110	95
80	93	85	97	90	94
88	93	90	93	99	93
89	94	92	90	107	94
89	94	94	86	106	94
84	96	87	88	113	94
75%	**0%**	**75%**	**50%**	**50%**	**100%**

Tabela 2 – Medida de desempenho do acesso aos eventos

geral, somente metade dos testes alcançam resultados ideais, o que representa um resultado levemente negativo e que necessita passar por modificações. Por fim, apresenta que 50% das buscas de 1.000 eventos com usuários A obtiveram resultados satisfatórios, estando abaixo dos 100 ms, enquanto que 100% das buscas com usuários NA alcançaram os mesmos objetivos, demonstrando que as buscas com usuários A precisam passar por melhorias, do mesmo modo que o mencionado acima e que as buscas com usuários NA apresentaram resultados muito positivos.

Do ponto de vista dos *stakeholders*, os resultados demonstram que os tempos de resposta obtidos possuem resultados ligeiramente positivos, no entanto, precisam de melhorias para alcançarem patamares superiores de qualidade, destacando-se que estas melhorias serão focadas principalmente nas consultas com quantidades menores de dados no contexto de busca por usuários NA. Terão uma alta prioridade também as buscas de uma quantidade média de eventos focando também em usuários NA e por fim, aprimorar a busca de eventos em grande quantidade por usuários A.

Além disso, para se obter resultados mais precisos e conclusivos, testes devem ser executados no *Web service* real da empresa ou em *Web service* que simule as situações reais da empresa, como largura de banda, latência da rede, possíveis interferências durante a requisição do serviço, entre outros, fatos que não ocorrem pelo fato dos testes estarem sendo executados em um *Web service* local.

O login foi avaliado com o intuito de averiguar o tempo médio estimado para o serviço autenticar o usuário no sistema. O teste foi desenvolvido conforme estabelecido em conjunto com os membros da empresa e apresentou resultado satisfatório, afirmando a hipótese de que o login pode ser efetuado aproximadamente na média de quinhentos milissegundos.

Os testes para avaliar se o tempo de resposta médio do serviço para efetuar a

confirmação de compra do usuário estivesse em uma média de três mil milissegundos, conforme reunião com os *stakeholders* da empresa. O teste foi desenvolvido conforme a *story* e apresentou bons resultados, pois a média foi relativamente inferior ao valor médio preestabelecido, ou seja, apresentou maior eficiência.

A Tabela 3 apresenta os resultados obtidos pelos dois testes anteriores mostrando o tempo médio de resposta do serviço para os testes de login e encaminhamento para etapa final da compra, bem como o desvio padrão e mediana para ambos os casos. Os valores dos tempos médios são apresentados em milissegundos.

Caso de teste	**Média**	**Desvio padrão**	**Mediana**
Login	509,75	3,536	509,5
Final da compra	2280,625	537,806	2190

Tabela 3 – Tempo de login e finalização da compra.

Os resultados mostram que a média de 509,75 obtida para o teste de login foi levemente negativa comparando-se a meta previamente estabelecida de 500 ms. O desvio padrão ficou em 3,536 e a mediana em 509,5. A proximidade entre média e mediana e o baixo valor do desvio padrão mostram que o serviço de login possui uma ótima estabilidade, trazendo confiança para o sistema. Já a média de 2.280,625 obtida para o teste de finalização de compra apresentou resultados muito satisfatórios, apresentando um valor bem menor que a meta preestabelecida de 3.000 ms. O desvio padrão foi de 537,806 e a mediana 2190, portanto, apesar da média e mediana apresentarem valores relativamente próximos, o desvio padrão representa aproximadamente 25% do valor total da média, mostrando ser um serviço instável de muita oscilação.

Do ponto de vista da equipe da empresa, os resultados mostram que o tempo de login está em concordância com a meta preestabelecida de 500 ms e os nove milissegundos que ultrapassaram o limite proposto pelos *stakeholders* pode ser ignorado, pois na prática essa diferença seria imperceptível do ponto de vista do usuário, e para o tempo de finalização de compra, os resultados surpreenderam a equipe, apontando valores muito menores que o esperado, no entanto, a oscilação de mais de um segundo entre as requisições do serviço serão levadas em consideração e melhorias serão aplicadas ao serviço, destacando-se aqui que o sistema ainda depende do tempo de resposta do sistema terceirizado para a otimização destes resultados. Além disso, estes testes seguem o mesmo princípio do teste de acesso de eventos quanto ao ambiente para aplicação dos testes.

Os testes de desempenho do login e finalização da compra foram desenvolvidos para estabelecer a qualidade do serviço com base nos tempos de acesso preestabelecidos juntamente com a equipe da empresa. Os valores obtidos foram próximos aos preestabelecidos na maioria

dos testes, com destaque para os testes de finalização de compra que obtiveram resultados relativamente menores do que os parâmetros modelo, consequentemente mais eficientes do que previamente estipulado. Os valores apresentados na Tabela 4 representam os resultados dos testes aplicados com o intuito de verificar o tempo de resposta do serviço para autenticar o usuário no sistema e o tempo de resposta do serviço para efetuar a confirmação de compra do usuário, sendo os valores apresentados em milissegundos e organizados pela categoria do teste aplicado. Além disso, as porcentagens apresentadas representam a quantidade de testes que alcançaram os valores preestabelecidos.

Caso de teste	**Testes**								**Desempenho**
Login	507	510	513	504	515	512	509	508	**0%**
Final da compra	3206	2444	1872	2059	1750	1730	2863	2321	**87,5%**

Tabela 4 – Medida de desempenho do login e da finalização da compra

Os resultados mostram que 0% das tentativas de login alcançaram os 500 ms propostos pelos *stakeholders*, sendo portanto, resultados extremamente negativos do ponto de vista estatístico. Nos testes de finalização de compra foram alcançados 87,5%, sendo um resultado ligeiramente positivo do serviço.

Do ponto de vista dos *stakeholders*, os resultados mostram que o serviço de login do sistema possui uma boa estabilidade quanto ao tempo de resposta e modificações no código poderão ocorrer para que este valor possa ser reduzido com o intuito de alcançar a meta dos 500 ms. Quanto a finalização de compra, o desvio padrão será levado em conta e verificações na estrutura de funcionamento do serviço serão efetuadas e terão uma alta prioridade na lista de modificações.

4.2 INTEGRIDADE

Os testes para verificar se o usuário somente poderia efetuar a compra do ingresso se estiver autenticado foram executados conforme estabelecido na *story*. Além disso, todos os testes obtiveram resultados satisfatórios, confirmando a integridade do *Web service* no quesito. O teste de *token* válido passou no teste obtendo-se o código de retorno *status code* 200, confirmando a autorização de compra pelo usuário. O teste *token* inválido também passou no teste obtendo-se o código de retorno *status code* 401, mostrando que o usuário não possuía

as credencias com informação de autenticação válidas para o serviço. Por fim, o teste sem um *token* de acesso também apresentou resultados positivos, gerando como saída o código de retorno *status code* 401: não autorizado.

Os testes que tinham por objetivo checar a geração da URL da etapa final da compra foram desenvolvidos em concordância com a hipótese estabelecida em conversa com os membros da equipe da empresa. Além disso, os testes foram desenvolvidos para a geração dinâmica de dez, cinquenta e cem compras no sistema, obtendo assim suas respectivas confirmações de compra, as quais apresentaram a URL da empresa terceirizada, responsável pela etapa final da compra. Os resultados apresentados na Tabela 5 mostram a razão entre o número de URLs geradas corretamente e o número de tentativas reproduzidas para o teste, apresentando os dados organizados pela quantidade de compras geradas dinamicamente.

Compras	**Integridade**
10	100%
50	100%
100	100%

Tabela 5 – Média de integridade da geração de URLs.

Os resultados mostram que as URLs foram geradas conforme proposto pela *story*. Ainda, relata que todas as URLs, nos três testes desenvolvidos, possuem sua estrutura em concordância com o modelo apresentado pela equipe da empresa, confirmando a integridade do *Web service* neste quesito. Com o intuito de coletar resultados mais precisos, testes com quantidades maiores de compras geradas dinamicamente poderão ser desenvolvidos.

4.3 SEGURANÇA

Os testes para verificar se usuários cadastrados no sistema somente teriam acesso aos seus próprios dados foram desenvolvidos de acordo com o proposto pela *story* e os resultados obtidos foram negativos. Foram realizados dois testes para a averiguação do caso: captura do identificador do usuário e utilização da autenticação de um usuário *user 1* para modificar um usuário *user 2*, gerando um código de retorno 401.

O primeiro teste foi executado e o serviço passou no teste apresentando o *status code* 200 como código de retorno e retornando o identificador do usuário como resposta do serviço,

enquanto que o segundo teste foi executado e o serviço falhou no teste apresentando também o *status code* 200 como código de retorno, quando deveria ter apresentado o código 401, como proposto. Além disso, a requisição PUT aplicada no teste foi realizada e os dados de *user 2* foram alterados. Destaca-se ainda que apesar da coleta do identificador do usuário ter sido efetuada com sucesso, ela foi realizada por meio de credenciais de autenticação de um usuário comum, portanto, apresenta também falha de segurança, permitindo que usuários sem privilégio de administrador do sistema possam acessar tais informações.

A *story* de alteração dos dados do evento ser unicamente efetuada pelo promotor do evento foi desenvolvida conforme especificado e os resultados obtidos não atingiram as expectativas. Um único teste foi desenvolvido utilizando as credenciais de um usuário *user 1* para com o intuito de alterar algum dado do evento promovido por um usuário *user 2*, e o serviço deveria apresentar como retorno o código *status code* 401, como proposto. O teste foi executado e o serviço falhou no teste apresentando como código de retorno o *status code* 200 e permitiu que o evento promovido por *user 2* fosse alterado por *user 1*.

Os testes com o intuito de verificar o acesso de um usuário aos ingressos de outro usuário foram desenvolvidos e apresentaram resultados fracos. Dois testes foram realizados para verificar a *story*: acessar os ingressos de *user 2* utilizando as credenciais de autorização de *user 1* deveria gerar um retorno *status code* 401 e acessar os ingressos de um usuário autenticado utilizando um usuário não autenticado no sistema deveria apresentar *status code* 401 como retorno.

O primeiro teste foi executado e o serviço falhou no teste apresentando o *status code* 200 como código de retorno e exibindo os ingressos de *user 1* na resposta de retorno ao invés de exibir os ingressos de *user 2*. Deste modo, notou-se que o serviço não é validado pela rota de requisição, mas pelas credenciais de autorização enviadas pelo requisitor, tornando indiferente o dado apresentado na rota desde que dentro dos parâmetros especificados. O segundo teste, por outro lado, foi executado e o serviço passou no teste apresentando o *status code* 401 como código de retorno, impedindo o requisitor do serviço de acessar os ingressos de um usuário autenticado.

Os testes para verificar o acesso de usuários as transações de outro usuário foram desenvolvidas e os resultados obtidos não alcançaram a meta preestabelecida pela *story*. Para a avaliação do caso foram desenvolvidos dois testes: acessar as transações de *user 2* utilizando as credenciais de autorização de *user 1* deveria gerar um retorno *status code* 401 e acessar as transações de um usuário autenticado utilizando um usuário não autenticado no sistema deveria apresentar *status code* 401 como retorno.

O primeiro teste foi executado e o serviço falhou no teste apresentando o *status code*

200 como código de retorno e exibindo os ingressos de *user 2* na resposta de retorno, enquanto que o segundo teste foi executado e o serviço passou no teste impedindo o requisitor do serviço de acessar as transações de um usuário autenticado, bem como apresentou o código de retorno, *status code* 401.

5 CONSIDERAÇÕES FINAIS

Neste documento foi mostrado a aplicação do *framework* REST Assured sobre um *Web service RESTful* de uma empresa real com o intuito de verificar a aplicabilidade da ferramenta de teste para este tipo de serviço. As qualidades de serviço abordadas na literatura proveram suporte aos *stakeholders*, esclarecendo alguns tópicos que ainda causavam dúvida e oferecendo flexibilidade na escolha de quais requisitos seriam abordados no trabalho. Além disso, adicionou qualidade ao produto da empresa que continuará a aplicação destas métricas de qualidade em seus produtos futuros.

As hipóteses levantadas juntamente com os membros de equipe da empresa responsável pelo produto, que vivenciam o cotidiano da aplicação do produto, ficou de acordo com as normativas do BDD e possibilitou que cenários mais concretos fossem apresentados.

A construção dos *Mock Objects* foi concluída com sucesso e foi essencial na aplicação dos testes sobre os serviços do *Web service*. Ainda, o desenvolvimento dos testes abrangendo pequenas funções de cada serviço, bem como organizando-os em estruturas modulares facilitou o desenvolvimento e aplicação dos testes em cada requisito abordado neste documento.

A aplicação dos testes utilizando o REST Assured possibilitou que diversas partes do *Web service* fossem avaliadas, apresentando não só resultados positivos comprovando a qualidade dos serviços, como também encontrou diversas falhas no sistema, que não tinham sido detectadas até o momento, levando-se em conta que a empresa já vinha utilizando alguns *frameworks* de teste de forma randômica, isto é, somente quando surgia uma dúvida grande o suficiente para ser efetuada a averiguação do caso.

O método do BDD não foi efetivamente aplicado neste trabalho, vide o fato de que o método requer o desenvolvimento dos testes seguidos do código de produção para passar naquele teste. No entanto, permitiu que o *framework* fosse aplicado sobre o sistema detectando diversas falhas em códigos de produção em seus respectivos *scenarios*. Desta forma, permitiu que os *stakeholders* percebessem a importância da aplicação de testes sobre o sistema e pudessem continuar a aplicação do método a partir deste momento, bem como começar a aplicação do BDD dos próximos serviços a serem implementados desde o início do desenvolvimento.

O REST Assured possibilitou a aplicação de testes sobre APIs independentemente de linguagem de programação, pois os testes podem ser codificados em projeto separado da API. Além disso, o desenvolvimento dos testes em linguagem de alto nível, evitando as interfaces gráficas existentes em ferramentas de testes presentes no mercado, ofereceu maior liberdade no desenvolvimento dos testes, criando assim testes mais específicos, que focavam exatamente nas necessidades apontadas pelos *stakeholders*. Diante disso, REST Assured apresentou-se como uma boa solução no desenvolvimento e aplicação de testes em *Web services RESTful*.

5.1 TRABALHOS FUTUROS

Os testes apresentados neste documento supriram algumas das necessidades da empresa nos requisitos abordados e proveram informação suficiente para apontar algumas considerações no trabalho. No entanto, muitos outros testes podem ser aplicados não só testando os mesmos serviços mencionados, bem como todas as métricas de QoS para gerar um produto final de qualidade.

O REST Assured apresenta diversas funcionalidades, abrangendo as mais variadas metodologias de desenvolvimento, que necessitam de maior tempo para estudo e compreensão, e que podem ser abordadas futuramente em novos projetos.

REFERÊNCIAS

ALBRESHNE, A.; FUHRER, P.; PASQUIER, J. Web services technologies: State of the art. Working Paper. 2009.

ANTE MERIDIEM DESIGN. **10 benefits of quality web design**. 2015. Acessado em: 22 Set. 2015. Disponível em: <http://www.antemeridiemdesign.com/articles/10-benefits-of-quality-web-design.php>.

BECK, K. **Test-driven development : by example**. Boston, San Francisco, Paris: Addison-Wesley, 2003. (The Addison-Wesley signature series). 12e tirage 2008. ISBN 0-321-14653-0. Disponível em: <http://opac.inria.fr/record=b1126571>.

BEIZER, B. **Software Testing Techniques (2Nd Ed.)**. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0.

BENTLEY, J. E.; BANK, W. Software testing fundamentals—concepts, roles, and terminology. In: **Proceedings of the Thirtieth Annual SAS® Users Group International Conference**. Charlotte, NC: SAS Institute Inc., 2005. Disponível em: <http://www2.sas.com/proceedings/sugi30/141-30.pdf>.

BOłOZ, D. M. **Improving use of behaviour-driven development in websphere commerce projects**. Maio 2014. Diploma de graduação.

BOZKURT, M. **Automated Realistic Test Input Generation and Cost Reduction in Service-centric System Testing**. Tese (Doutorado) — University College London (University of London), Londo, UK, Junho 2013. Disponível em: <http://www0.cs.ucl.ac.uk/staff/mharman/mustafa-phd.pdf>. Acesso em: 02 de maio de 2016.

CAELUM. O que é java. In: **Java e Orientação a Objetos**. Caelum: ensino e inovação, 2014. cap. 2, p. 3–8. Disponível em: <https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf>. Acesso em: 18 de abril de 2016.

CASLINO. Blog, **Software testing**. Optimus Information Inc., 2014. Acessado em: 24 Set. 2015. Disponível em: <http://www.optimusinfo.com/blog/5-advantages-automated-software-testing/>.

CHAPPELL, D.; JEWELL, T. **Java Web Services**. 1. ed. [S.l.]: O'Reilly, 2002. 276 p. ISBN 0-596-00269-6.

CHELIMSKY, D. et al. **The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends**. 1st. ed. [S.l.]: Pragmatic Bookshelf, 2010. ISBN 1934356379, 9781934356371.

COWAN, J. **RESTful Web Services**. [s.n.], 2005. 23 - 31 p. Disponível em: <http://home.ccil.org/~cowan/restws.pdf>.

DELSHAD, P. **Behavior Driven Development in a Large-Scale Application: Evaluation of Usage for Developing IFS Applications**. 87 p. Dissertação (Mestrado) — Linköping University, Software and Systems, 2016.

DOHMKE, T. **Test-driven development of embedded control systems: application in an automotive collision prevention system**. Tese (Doutorado) — University of Glasgow, Glasgow, UK, Fevereiro 2008.

DUPAUL, N. **Software Testing Tools and Techniques**. 2015. Acessado em: 17 Out. 2015. Disponível em: <http://www.veracode.co.uk/security/software-testing-tools>.

ENES, P. **Build and Release Management: supporting development of accelerator control software at CERN**. Dissertação (Mestrado) — Norwegian University of Science and Technology, 2007.

FERNANDES, P. F.; MEDEIROS, M. **Quality Assessment for Geographic Web Services**. Dissertação (Mestrado) — Universidade Técnica de Lisboa, Lisboa, PT, Outubro 2009.

FIELDING, E. R.; RESCHKE, E. J. **Hypertext Transfer Protocol (HTTP/1.1): Authentication**. [S.l.]: IETF, 2014. IETF 7235.

FIELDING, E. R.; RESCHKE, E. J. **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content**. [S.l.]: IETF, 2014. IETF 7231.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado), 2000. AAI9980887.

FOFUNG, T. D. **Formal Specification Driven Development**. Dissertação (Mestrado) — Kennesaw State University, 2015.

GARCÍA, B.; DUEÑAS, J. C. Automated functional testing based on the navigation of web applications. In: **Proceedings 7th International Workshop on Automated Specification and Verification of Web Systems, WWV 2011, Reykjavik, Iceland, 9th June 2011.** [s.n.], 2011. p. 49–65. Disponível em: <http://dx.doi.org/10.4204/EPTCS.61.4>.

GEORGE, B. **Analysis and Quantification of Test Driven Development Approach**. Dissertação (Mestrado) — North Carolina State University, Raleigh, US, 2002.

GHANAKOTA, G. **TESTING FRAMEWORKS**. University of Colorado Boulder, 2012. Acessado em: 17 Out. 2015. Disponível em: <http://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/ghanakotagayatri.pdf>.

GODEFROID, P. et al. Automating software testing using program analysis. **Software, IEEE**, v. 25, n. 5, p. 30–37, Sept 2008. ISSN 0740-7459.

GRACIAS, S. **Automation Test Frameworks**. University of Colorado Boulder, 2010. Acessado em: 17 Out. 2015. Disponível em: <http://www.cs.colorado.edu/~kena/classes/5828/s10/presentations/automation_test_frameworks.pdf>.

GRATER, M. T. **Benefits of Using Automated Software Testing Tools to Achieve Software Quality Assurance**. Portland, OR, June 2005. Disponível em: <https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/7817/2005-grater.pdf>.

HALEBY, J. **REST Assured 2.0 – Testing your REST services is easier than ever**. 2013. Acessado em: 20 Out. 2015. Disponível em: <http://www.jayway.com/2013/11/29/rest-assured-2-0-testing-your-rest-services-is-easier-than-ever/>.

HALEBY, J. **REST Assured GettingStarted**. 2016. Disponível em: <https://github.com/rest-assured/rest-assured/wiki/GettingStarted>. Acesso em: 23 de junho de 2016.

HALEBY, J. **REST Assured Wiki Usage**. 2016. Disponível em: <https://github.com/rest-assured/rest-assured/wiki/Usage>. Acesso em: 23 de junho de 2016.

HALFOND, W. G. J.; ANAND, S.; ORSO, A. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In: **Proceedings of the International Symposium on Software Testing and Analysis**. Chicago, Illinois, USA: [s.n.], 2009. p. 285–296.

HAYES, L. G. **The Automated Testing Handbook**. Software Testing Institute, 2004. ISBN 9780970746504. Disponível em: <http://www.softwaretestpro.com/itemassets/4772/automatedtestinghandbook.pdf>.

HENDRICKSON, E. **Repeatability is Overrated**. 2006. Acessado em: 23 Set. 2015. Disponível em: <http://testobsessed.com/2006/11/repeatability-is-overrated/>.

HILARI, M. O. **Monitoring the Quality of Service to support the Service Based System lifecycle**. Tese (PhD thesis) — Universitat Politècnica de Catalunya, Janeiro 2015.

HILTON, R. **Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects**. Dissertação (Mestrado) — Regis University, Denver, Colorado - US, Dezembro 2009.

ISO/IEC. **ISO 8402:1994 - Quality management and quality assurance – Vocabulary**. [S.l.]: ISO/IEC, 1994.

ISO/IEC. **ISO 9126:2001 - Software engineering – Product quality**. [S.l.]: ISO/IEC, 2001.

ISO/IEC. **ISO 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**. [S.l.]: ISO/IEC, 2011.

JASEK, P. **Test-Driven Development: 15 years later**. Dissertação (Mestrado) — Aalborg University, Aalborg, DK, Setembro 2014.

KEARNEY, A.; RANDS, M. **Overview of Software Testing**. 2015. Acessado em: 19 Set. 2015. Disponível em: <http://www.bcs.org/content/conWebDoc/7942>.

KEEN, M. et al. IBM Redpaper, IBM Software Group, **Developing Web Services Applications**. 2012. Acessado em: 01 Nov. 2015. Disponível em: <http://www.redbooks.ibm.com/abstracts/redp4884.html?Open>.

KELLY, M. Choosing a test automation framework. In: **IBM developerWorks®**. IBM developerWorks®, 2003. Disponível em: <http://www.ibm.com/developerworks/rational/library/591.html>.

KHANDKAR, S. H. et al. Fitclipse: A tool for executable acceptance test driven development. In: ABRAHAMSSON, P.; MARCHESI, M.; MAURER, F. (Ed.). **Agile Processes in Software Engineering and Extreme Programming: 10th International Conference, XP 2009, Pula, Sardinia, Italy, May 25-29, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. (Lecture Notes in Business Information Processing, v. 31), p. 259–260. ISBN 978-3-642-01853-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-01853-4_58>.

KOPS, M. **Testing RESTful Web Services made easy using the REST-assured Framework**. 2011. Acessado em: 20 Out. 2015. Disponível em: <http://www.hascode.com/2011/10/testing-restful-web-services-made-easy-using-the-rest-assured-framework/>.

KOSKELA, L. **Test Driven: Practical Tdd and Acceptance Tdd for Java Developers**. Greenwich, CT, USA: Manning Publications Co., 2007. ISBN 9781932394856.

KOUDELIA, N. **Acceptance Test-Driven Development**. Dissertação (Mestrado) — UNIVERSITY OF JYVÄSKYLÄ, Dezembro 2011.

KREGER, H. Technical report, IBM Software Group, **Web Services Conceptual Architecture (WSCA 1.0)**. Maio 2001. Acessado em: 01 Nov. 2015. Disponível em: <http://users.cs.uoi.gr/˜pitoura/courses/ds04_gr/webt.pdf>.

LAUKKANEN, P. **Data-driven and keyword-driven test automation frameworks**. Tese (Doutorado) — Helsinki University of Technology, 2006.

LIJUAN, W.; YUE, Z.; HONGFENG, H. Genetic algorithms and its application in software test data generation. In: **Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on**. [S.l.: s.n.], 2012. v. 2, p. 617–620.

MANDEL, L. Describe rest web services with wsdl 2.0: a how-to guide. In: **IBM®developerWorks®**. IBM, 2008. Disponível em: <http://www.ibm.com/developerworks/library/ws-restwsdl/ws-restwsdl-pdf.pdf>. Acesso em: 02 de maio de 2016.

MANI, A.; NAGARAJAN, A. Understanding quality of service for web services: Improving the performance of your web services. In: **IBM®developerWorks®**. IBM, 2002. Disponível em: <http://www.ibm.com/developerworks/library/ws-quality/ws-quality-pdf.pdf>. Acesso em: 06 de maio de 2016.

MBURUGU, C. **The benefits of website usability testing**. 2012. Acessado em: 19 Set. 2015. Disponível em: <http://www.productivedreams.com/the-benefits-of-website-usability-testing/>.

MELNIK, G. I. **Empirical Analyses of Executable Acceptance Test Driven Development**. Tese (Doutorado), Calgary, Alta., Canada, Canada, 2007. AAINR33806.

MäKINEN, S. **Driving Software Quality and Structuring Work Through Test-Driven Development**. Dissertação (Mestrado) — UNIVERSITY OF HELSINKI, Setembro 2012.

NEČAS, I. **BDD as a Specification and QA Instrument**. Dissertação (Mestrado) — Masaryk University, Faculty of Informatics, Brno, 2011. Disponível em: <http://theses.cz/id/4l7mxn/>.

NILSSON, N. **Test-Driven Development in Clojure: An analysis of how differences between Clojure and Java affects unit testing and design patterns**. Dissertação (Mestrado) — KTH Royal Institute of Technology, Stockholm, SE, Março 2015.

ORACLE DOCS. **Lesson: Object-Oriented Programming Concepts**. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>. Acesso em: 18 de abril de 2016.

ORACLE DOCS. **Polymorphism**. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html>. Acesso em: 18 de abril de 2016.

PALANI, G. S. Summary of web application testing methodologies and tools. In: **IBM developerWorks®**. IBM, 2011. Disponível em: <http://www.ibm.com/developerworks/library/wa-webapptesting/wa-webapptesting-pdf.pdf>.

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. Restful web services vs. big web services: Making the right architectural decision. In: **17th International World Wide Web Conference (WWW2008)**. Beijing, China: [s.n.], 2008. p. 805–814. Disponível em: <http://www2008.org/>.

PERRY, J. S. Introduction to java programming, part 1: Java language basics. In: **IBM®developerWorks®**. IBM, 2015. Disponível em: <http://www.ibm.com/developerworks/java/tutorials/j-introtojava1/j-introtojava1-pdf.pdf>. Acesso em: 18 de abril de 2016.

PHAM, Q.; MALIK, T.; FOSTER, I. Using provenance for repeatability. In: **Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance**. Berkeley, CA: USENIX, 2013. Disponível em: <https://www.usenix.org/conference/tapp13/using-provenance-repeatability>.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN 0073375977, 9780073375977.

RAJENDRAN, T.; BALASUBRAMANIE, P. Analysis on the study of qos-aware web services discovery. **CoRR**, abs/0912.3965, 2009. Disponível em: <http://arxiv.org/abs/0912.3965>.

ÁRIAS, J. C. G. **Teste de aplicações baseado em análise de instâncias de dados alternativas**. Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, PR, 2011.

RICHARDSON, L.; RUBY, S. **Restful Web Services**. First. [S.l.]: O'Reilly, 2007. 79-105 p. ISBN 9780596529260.

RUTH, M. E. **Automating Regression Test Selection for Web Services**. Tese (Doutorado) — University of New Orleans, Agosto 2007.

SAMPATH, S. **Cost-effective Techniques for User-session-based Testing of Web Applications**. Tese (Doutorado), Newark, DE, USA, 2006. AAI3220722.

SATHISH, C. G. **Software testing**. Visvesvaraya Technological University, 2012. Acessado em: 24 Set. 2015. Disponível em: <http://elearning.vtu.ac.in/12/enotes/Soft_Test/Unit1-SV.pdf>.

SELLEO. **5 advantages of automated testing vs. manual testing**. SELLEO, 2015. Acessado em: 24 Set. 2015. Disponível em: <http://selleo.com/blog/software-outsourcing/the-value-of-automated-testing/3/>.

SOLINO, A. L. da S. **Teste baseado em defeitos para web services**. Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, PR, 2008.

SOROOR, J.; TAROKH, M. J. Developing the next generation of the web and employing its potentials for coordinating the supply chain processes in a mobile real-time manner. **International Journal of Information Technology**, v. 12, n. 8, p. 1–40, 2006.

SRIVASTAVA, P. R.; KIM, T. hoon. Application of genetic algorithm in software testing. In: **International Journal of Software Engineering and Its Applications**. [S.l.: s.n.], 2009. v. 3, p. 87 – 96.

THE APACHE SOFTWARE FOUNDATION. **Introduction**. 2016. Disponível em: <https://maven.apache.org/what-is-maven.html>. Acesso em: 27 de abril de 2016.

TOMB, A.; FLANAGAN, C. Detecting inconsistencies via universal reachability analysis. In: **Proceedings of the 2012 International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2012. (ISSTA 2012), p. 287–297. ISBN 978-1-4503-1454-1. Disponível em: <http://doi.acm.org/10.1145/2338965.2336788>.

VANCE, A.; CICKOVSKI, T. A case study on developing a classroom web application using behavior-driven development. **American journal of undergraduate research**, v. 11, n. 3–4, p. 9–17, Maio 2012.

W3C. **Web Services Description Language (WSDL) 1.1**. 2001. Disponível em: <https://www.w3.org/TR/wsdl>. Acesso em: 02 de maio de 2016.

W3C. **QoS for Web Services: Requirements and Possible Approaches**. 2003. Disponível em: <http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/>. Acesso em: 06 de maio de 2016.

W3C. **Extensible Markup Language (XML) 1.1 (Second Edition)**. 2006. Disponível em: <https://www.w3.org/TR/xml11/>. Acesso em: 03 de maio de 2016.

W3C. **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)**. 2007. Disponível em: <https://www.w3.org/TR/soap12/>. Acesso em: 03 de maio de 2016.

WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. **REST in Practice: Hypermedia and Systems Architecture**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2010. ISBN 0596805829, 9780596805821.

WIKIPéDIA. **Visão geral dos atributos de qualidade da Norma ISO 9126**. 2016. Disponível em: <https://pt.wikipedia.org/wiki/ISO/IEC_9126>. Acesso em: 06 de maio de 2016.

WILLIAMS, L. **Agile/Automated Testing**. North Carolina State University, 2004. Acessado em: 24 Set. 2015. Disponível em: <http://agile.csc.ncsu.edu/SEMaterials/AgileTesting.pdf>.