

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Rodrigo Bittencourt de Lima

**TESTES DE INTEGRAÇÃO AUTOMATIZADOS SOBRE
API REST DO SOFTWARE HAWA**

Florianópolis

2019

Rodrigo Bittencourt de Lima

**TESTES DE INTEGRAÇÃO AUTOMATIZADOS SOBRE
API REST DO SOFTWARE HAWA**

Tese submetida à Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador: Prof. Dr. Martin Augusto Gagliotti Vigil
Coorientador: Prof. Dr. Leandro José Komosinski

Florianópolis

2019

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

Lima, Rodrigo Bittencourt de
Testes de integração automatizados sobre API REST do
software Hawa / Rodrigo Bittencourt de Lima ; orientador,
Martin Augusto Gagliotti Vigil, coorientador, Leandro José
Komosinski, 2019.
140 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Sistema de Informação, Florianópolis, 2019.

Inclui referências.

1. Sistema de Informação. 2. testes automatizados. 3.
hawa. 4. api rest. 5. certificados digitais. I. Vigil,
Martin Augusto Gagliotti. II. Komosinski, Leandro José.
III. Universidade Federal de Santa Catarina. Graduação em
Sistema de Informação. IV. Título.

Rodrigo Bittencourt de Lima

**TESTES DE INTEGRAÇÃO AUTOMATIZADOS SOBRE
API REST DO SOFTWARE HAWA**

Esta Tese foi julgada aprovada para a obtenção do Título de “Bacharel em Sistemas de Informação”, e aprovada em sua forma final pela Sistemas de Informação.

Florianópolis, 30 de novembro 2019.

Prof. Dr. Cristian Koliver
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Martin Augusto Gagliotti Vigil
Orientador

Prof. Dr. Leandro José Komosinski
Coorientador

Lucas Machado da Palma

Pablo Rinco Montezano

Dedico este trabalho à minha mãe, que sempre deu o seu melhor para me oferecer estudo de qualidade e me possibilitou ingressar em uma universidade federal.

AGRADECIMENTOS

Agradeço primeiramente à minha mãe, que me possibilitou estudar em bons colégios durante toda a vida, adquirindo os conhecimentos necessários para ingressar em uma universidade federal. À familiares e amigos próximos que me incentivaram em momentos difíceis. Ao meu orientador Dr. Martin Vígil, por toda a paciência e compreensão que teve comigo, e por todo o conhecimento compartilhado. Ao meu coorientador Dr. Leandro Komosinski, por toda a experiência e conhecimentos compartilhados. Ao LabSEC, por me oferecer a oportunidade e me auxiliar em todas as etapas do presente trabalho, em especial ao Pablo Montezano por toda a ajuda e por todo o conhecimento compartilhado.

*Deixarás de temer quando deixares de ter
esperança.*

Sêneca

RESUMO

Com a crescente adoção de metodologias ágeis no desenvolvimento de software, muitos processos precisaram ser adaptados à nova realidade. O processo de garantia de qualidade de software precisou ser evoluído para gerar uma maior eficiência durante os ciclos iterativos de entrega utilizados nas metodologias ágeis. Testes manuais se tornaram ainda mais custosos devido ao excesso de testes de regressão e ao retrabalho. Com isso, cada vez mais a automatização de testes ganhou importância, tornando-se fator decisivo para a eficiência do processo de desenvolvimento de software como um todo. O objetivo do presente trabalho é garantir o correto funcionamento da API REST do software Hawa, sendo este propriedade do Laboratório de Segurança em Computação da Universidade Federal de Santa Catarina e cuja finalidade é a emissão de certificados digitais a usuários finais. O Hawa é utilizado pelas entidades de hierarquia mais baixa na cadeia de entidades certificadoras da Infraestrutura de Chaves Públicas Brasileira. Para alcançar o objetivo proposto, foram implementados e executados testes automatizados de integração sobre a API REST do Hawa, utilizando ferramentas e tecnologias atuais. Como resultados deste trabalho, verificou-se que a camada pública do software (sua API REST) se comporta conforme o esperado. Além disso, os testes automatizados de integração implementados durante a execução deste trabalho foram entregues à equipe responsável pelo Hawa para uso futuro. Documentos orientando a utilização dos testes e maneiras de incluí-los em futuras pipelines de validação foram escritos e disponibilizados para os responsáveis pelo software.

Palavras-chave: Metodologias ágeis, Testes de software, Testes automatizados, API, Rest.

ABSTRACT

With the increasing adoption of agile methodologies in software development, many processes needed to be adapted to the new reality. The software quality assurance process needed to be evolved to generate greater efficiency during the iterative delivery cycles used in agile methodologies. Manual testing has become even more costly due to excessive regression testing and rework. As a result, testing automation has become increasingly important, becoming a decisive factor for the efficiency of the software development process as a whole. The objective of the present work is to ensure the correct functioning of the Hawa Software REST API, which belongs to the Computer Security Laboratory of the Federal University of Santa Catarina and has been created to issue digital certificates to end users. Hawa is used by the lower hierarchy entities in the Brazilian Public Key Infrastructure. To achieve the proposed goal, automated integration tests on Hawa's Rest API were implemented and executed using current tools and technologies. As a result of this work, it was found that the public software layer (its Rest API) behaves as expected. In addition, the automated integration tests implemented during the execution of this work were delivered to the Hawa team for future use. Documents guiding the use of the tests and ways to include them in future validation pipelines have been written and made available to those responsible for the software. **Keywords:** Agile Methodologies, Software Testing, Automated Testing, API, Rest.

LISTA DE FIGURAS

Figura 1	Etapas do modelo cascata. Fonte: Engenharia de Software - 8ª Edição	28
Figura 2	Custo das mudanças de requisitos nas etapas do modelo cascata. Fonte: Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software.....	33
Figura 3	Custo das mudanças de requisitos nos modelos ágeis de desenvolvimento. Fonte: Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software	33
Figura 4	Notação Gherkin. Fonte: Elaboração própria	43
Figura 5	Exemplo de comunicação cliente-servidor stateless. Fonte: RESTful Web services: The basics	46
Figura 6	Formato XML. Fonte: Elaboração própria	48
Figura 7	Formato JSON. Fonte: Elaboração própria	48
Figura 8	Hierarquia ICP-Brasil. Fonte: site Benefícios e Aplicações da Certificação Digital.....	49
Figura 9	Exemplo de código de teste implementado com Rest-Assured. Fonte: site oficial da ferramenta	56
Figura 10	Funcionamento do Postman. Fonte: documentação oficial da ferramenta.....	57
Figura 11	Layout do Postman. Fonte: documentação oficial da ferramenta	58
Figura 12	Adição de scripts à requisições no Postman. Fonte: documentação oficial da ferramenta.....	60
Figura 13	Script de teste. Fonte: Elaboração própria	61
Figura 14	Requisição de teste. Fonte: Elaboração própria	61

LISTA DE ABREVIATURAS E SIGLAS

LabSEC	Laboratório de Segurança em Computação
ICP	Infraestrutura de Chaves Públicas
API	Application Programming Interface
REST	Representational State Transfer
JSON	JavaScript Object Notation
UFSC	Universidade Federal de Santa Catarina
BDD	Behavior Driven Development
XP	EXtreme Programming
IEEE	Institute of Electrical and Electronic Engineers
SQA	Software quality assurance
TDD	Test Driven Development
SBE	Specification by example
HTTP	Hypertext Transfer Protocol
XML	Extensible Markup Language
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
URI	Uniform Resource Identifier
RFC	Request for Comments
XHTML	EXtensible HyperText Markup Language
ITI	Instituto Nacional de Tecnologia da Informação
AC	Autoridade Certificadora
AR	Autoridade de Registro
IDE	Integrated Development Environment
HTML	Hypertext Markup Language
E2E	End-to-end

SUMÁRIO

1 INTRODUÇÃO	23
1.1 OBJETIVOS	24
1.1.1 Objetivos específicos	24
1.2 METODOLOGIA	25
1.3 ORGANIZAÇÃO DO TRABALHO	26
2 CONCEITOS PRELIMINARES	27
2.1 METODOLOGIAS TRADICIONAIS DE DESENVOLVIMENTO DE SOFTWARE	27
2.2 METODOLOGIAS ÁGEIS DE DESENVOLVIMENTO DE SOFTWARE	30
2.3 QUALIDADE DE SOFTWARE	34
2.3.1 Testes de software	35
2.3.2 Testes automatizados de software	37
2.3.3 Norma IEEE 829	39
2.4 BEHAVIOR-DRIVEN DEVELOPMENT	41
2.4.1 Gherkin	43
2.5 API REST	44
2.5.1 Métodos HTTP	44
2.5.2 Stateless	45
2.5.3 URI	46
2.5.4 Envio de dados em formato XML, JSON, ou ambos	47
2.6 HAWA	48
3 PLANEJAMENTO	51
3.1 PLANO DE TESTE	51
3.2 ESPECIFICAÇÃO DOS CASOS DE TESTE	53
3.3 DEFINIÇÃO DE TECNOLOGIAS E FERRAMENTAS ...	53
3.3.1 Apache JMeter	53
3.3.2 Katalon Studio	54
3.3.3 SoapUI	54
3.3.4 Rest-Assured	55
3.3.5 Postman	56
3.3.6 Definição de ferramentas	57
4 IMPLEMENTAÇÃO	59
4.1 AMBIENTE DE TESTES	59
4.2 TESTES AUTOMATIZADOS COM POSTMAN	59
5 ANÁLISE DE RESULTADOS	63
5.1 RESULTADOS POSITIVOS	63

5.2 RESULTADOS NEGATIVOS	64
6 CONSIDERAÇÕES FINAIS.....	65
6.1 TRABALHOS FUTUROS	65
7 DIREITOS AUTORAIS.....	67
REFERÊNCIAS	69
APÊNDICE A - Plano de teste	75
APÊNDICE B - Casos de teste	83
APÊNDICE C - Código dos testes automatizados	95
APÊNDICE D - Guia para uso dos testes automatizados	111
APÊNDICE E - Guia para uso dos testes automatizados em pipelines de validação.....	125
APÊNDICE F - Artigo	133

1 INTRODUÇÃO

A indústria de software busca continuamente por modelos de desenvolvimento de software eficazes e cada vez mais eficientes. As metodologias ágeis de desenvolvimento se mostram eficientes e lucrativas, possuindo maior destaque na atualidade. As novas metodologias disputam espaço com os modelos tradicionais de desenvolvimento de software ainda utilizados (FADEL; SILVEIRA, 2010).

As metodologias ágeis, utilizando-se de ciclos curtos e iterativos de entrega, além de contato constante com o cliente, mostram-se efetivas na redução de custos. Para que as metodologias ágeis sejam capazes de aumentar a eficiência do processo de desenvolvimento de software, profissionais e atividades rotineiramente executadas durante esse processo precisam de adaptações. Quando pensamos no processo de garantia de qualidade de software e analisamos a evolução constante do mesmo, é evidente que trata-se de um dos processos que vêm sofrendo mudanças constantemente (SEMEDO, 2012).

Neste contexto, testes manuais de software se mostram caros e exaustivos. Isso porque são necessários muitos testes de regressão repetitivos e um conseqüente retrabalho. DELAMARO et al. (2016) afirmam que testes regressivos consistem em um tipo de teste cujo objetivo é validar que um software se mantém consistente após a adição de novas funcionalidades. No contexto das metodologias de desenvolvimento tradicionais, são testes executados geralmente na etapa de manutenção do software. Tornou-se necessário pensar em soluções para adaptar o processo de garantia de qualidade de software às metodologias de desenvolvimento emergentes. A utilização de testes automatizados de regressão, substituindo os testes manuais, mostrou-se uma solução viável e promissora. Por meio dessa estratégia de testes é possível evitar ciclos repetitivos e humanamente exaustivos de validação, reduzindo custos e evoluindo a qualidade dos softwares implementados (BERNARDO; KON, 2008).

Considerando todos os benefícios oriundos da automatização de testes, o presente trabalho surgiu da necessidade de garantia de qualidade automatizada sobre a API REST do software Hawa. Este produto pertence ao Instituto Nacional de Tecnologia da Informação e é desenvolvido pelo Laboratório de Segurança em Computação (LabSEC) da Universidade Federal de Santa Catarina (UFSC), e sua finalidade é a emissão de certificados digitais a usuários finais. O Hawa é utilizado pelas entidades pertencentes à cadeia certificadora da Infraestrutura de

Chaves Públicas Brasileira (ICP - Brasil), mais especificamente pelas de hierarquia mais baixa. A ICP - Brasil consiste em uma estrutura hierárquica que permite a emissão de certificados digitais para identificação virtual do cidadão por meio de uma cadeia de confiança (Instituto Nacional de Tecnologia da Informação - ITI, 2017).

A equipe de desenvolvimento do Hawa já faz o uso de testes automatizados para validação do produto, fato de extrema importância levando em consideração à quantidade restrita de profissionais envolvidos no projeto. Porém, a equipe dispõe de tempo suficiente apenas para a implementação de testes de unidade sobre o software. Testes de unidade consistem em scripts automatizados úteis na validação das menores partes de um software. Por exemplo, funções, procedimentos, métodos ou classes. Esse tipo de teste se limita a garantir que cada pequena parte de um programa funcione corretamente, mas não é capaz de validar o funcionamento de um software de forma integrada (DELAMARO et al., 2016).

O presente trabalho surgiu da necessidade por parte da equipe de profissionais responsáveis pela implementação do software, de garantir que a API REST pública existente no mesmo funcione corretamente e de forma integrada a cada nova alteração realizada no código do produto. Como a equipe é reduzida e o tempo limitado, utilizar apenas testes manuais durante essa validação é ineficiente e não é viável implementar uma suíte de testes automatizados de integração internamente à equipe. Devido a esses fatos, tornaram-se necessárias a modelagem e a implementação de uma suíte de testes automatizados de integração externamente à equipe, gerando a oportunidade de executar este trabalho.

1.1 OBJETIVOS

Prover uma suíte de testes automatizados de integração para a API REST pública do software Hawa.

1.1.1 Objetivos específicos

- Implementar uma suíte automatizada de testes de integração com base na documentação da API REST.
- Executar a suíte de testes e coletar os resultados.

- Disponibilizar para a equipe do Hawa um arquivo contendo a implementação dos casos de testes automatizados de integração, um documento com instruções para execução dos testes automatizados, e um documento com orientações para inclusão dos mesmos em pipelines de validação.

1.2 METODOLOGIA

A metodologia utilizada para execução do presente trabalho foi baseada na Norma IEEE 829, com algumas adaptações necessárias devido ao porte do projeto. A Norma IEEE consiste em um conjunto de orientações a serem utilizadas durante a execução do processo de teste de software. Essa norma orienta que o modelo proposto deve ser adaptado a realidade de cada projeto (CRESPO et al., 2004).

Levando em consideração os objetivos específicos, a ordem das atividades ocorrerá da seguinte maneira:

1. Buscar contato com os responsáveis pelo projeto Hawa com o intuito de conhecer o software, definir expectativas e resultados esperados.
2. Desenvolver um plano de testes para guiar a execução do trabalho.
3. Estudar a documentação da API Rest do Hawa, desenhar e validar os casos de teste necessários com base na mesma.
4. Estudar as ferramentas disponíveis no mercado, definindo as ideias para uso durante o presente trabalho.
5. Implementar, validar e executar a suíte de testes automatizados de integração.
6. Coletar os resultados e disponibilizá-los à equipe responsável pelo software.
7. Elaborar os documentos necessários para o completo entendimento dos artefatos construídos durante o trabalho, e realizar as entregas esperadas aos interessados.

Como artefato complementar à implementação da suíte de testes automatizados de integração, é necessário produzir uma monografia contendo todos os conceitos e atividades relevantes para este trabalho. Tal documento será elaborado paralelamente as etapas citadas acima, pois as informações necessárias para tal são obtidas ao longo do projeto.

1.3 ORGANIZAÇÃO DO TRABALHO

Os capítulos que seguem estão organizados da seguinte forma: no capítulo 2 são levantados os principais conceitos utilizados para justificar, planejar e implementar a suíte de casos de teste. O capítulo 3 discorre como é planejada a suíte de casos de teste automatizados. O capítulo 4 detalha o processo de implementação da mesma. O capítulo 5 traz a discussão dos resultados obtidos. Por fim, o capítulo 6 apresenta as considerações finais.

2 CONCEITOS PRELIMINARES

Este capítulo apresenta os conceitos necessários para que o leitor compreenda este trabalho. Inicia-se introduzindo os conceitos relacionados às metodologias tradicionais de desenvolvimento de software. Posteriormente e com intuito de comparação, são apresentados os conceitos das metodologias ágeis de desenvolvimento de software. Em seguida, apresentam-se os conceitos relacionados ao processo de garantia de qualidade de software. Após, a técnica de desenvolvimento de software conhecida como BDD é discutida. São abordados também os conceitos relacionados com API REST. Finalmente, conceitos relacionados ao software Hawa são elencados.

2.1 METODOLOGIAS TRADICIONAIS DE DESENVOLVIMENTO DE SOFTWARE

A realidade da indústria de desenvolvimento de software é dinâmica, mudando constantemente devido aos frequentes avanços tecnológicos. As metodologias tradicionais de desenvolvimento de software, também conhecidas como metodologias pesadas de desenvolvimento, ou ainda como metodologias orientadas à documentação, foram pensadas em um momento diferente do vivido pela engenharia de software atualmente.

Na época de surgimento das metodologias citadas, o acesso a computadores e a recursos para desenvolvimento de software era extremamente limitado. Conseqüentemente, o custo para realizar modificações e evoluções em um programa de computador era muito alto. É importante ressaltar também que naquele momento ferramentas de apoio ao desenvolvimento de software, como depuradores e analisadores de código, ainda não existiam. A estratégia adotada para tentar diminuir os custos de desenvolvimento era planejar e documentar ao máximo um software antes de sua implementação. Uma das mais conhecidas metodologias tradicionais de desenvolvimento, muito utilizada até os dias atuais, é o modelo Clássico (SOARES, 2004a).

O sequenciamento de processos é uma característica forte do modelo Clássico, muito conhecido como modelo Cascata. Isso fica claro quando se afirma que o planejamento e a documentação de um software devem ocorrer por completo antes de sua implementação, como citado anteriormente. É possível resumir os processos definidos pelo modelo

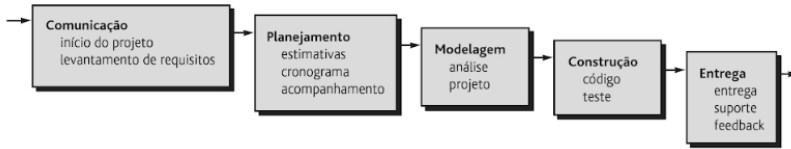


Figura 1 – Etapas do modelo cascata. Fonte: Engenharia de Software - 8ª Edição

Cascata em especificação de requisitos, planejamento e modelagem, implementação e implantação, e por fim manutenção do software, todos executados integralmente e nessa sequência. Este modelo é considerado o mais antigo, porém devido ao crescente avanço tecnológico, perde cada vez mais utilidade na indústria de desenvolvimento de software como consequência dos problemas que apresenta quando utilizado no contexto atual do mercado (PRESSMAN e MAXIM, 2016).

Assim como SOARES (2004a), PRESSMAN e MAXIM (2016) também consideram especificação de requisitos, planejamento e modelagem, implementação e implantação, e manutenção, as 5 etapas principais do processo de desenvolvimento de software proposto pelo modelo Cascata.

PRESSMAN e MAXIM (2016) conceituam o processo de especificação de requisitos como o responsável por definir as características de um software a ser desenvolvido, tanto requisitos como restrições. São necessárias constantes interações com os interessados no produto em análise para realizar tais definições. Já o processo de planejamento e modelagem do software envolve várias atividades, desde definição de estimativas e estabelecimento de prazos, até o projeto da solução que deve ser implementada para resolver os problemas apresentados pelo cliente. O processo de implementação e implantação pode ser visto como a codificação do software, validação e sua disponibilização aos interessados. Como processo final do modelo, a manutenção ocorre quando um software já em uso pelo cliente apresenta erros de funcionamento que necessitam de correção. As etapas do modelo Cascata são ilustradas na Figura 1.

SOARES (2004a) cita várias características do modelo Cascata que fazem com que o mesmo seja cada vez menos utilizado no mercado. Com os avanços tecnológicos, o processo de codificação de um software tornou-se muito mais barato. No contexto atual, o acesso a computadores e a ferramentas de suporte a codificação é praticamente

ilimitado. Tendo isso em vista, a inflexibilidade do modelo Cascata, definindo processos sequenciais e que devem ser executados integralmente, dificulta a realização de alterações e evoluções em um software, ações que se tornaram possíveis devido ao contexto tecnológico atual já comentado.

O autor afirma que, com as condições existentes, o modelo Cascata só deveria ser utilizado quando as características desejáveis para um software estejam muito claras. SOARES (2004a) afirma que é inviável especificar um software com total assertividade de detalhes antes do início da implementação do mesmo, e sugere a utilização de modelos incrementais de desenvolvimento de software, visando maior eficiência e lucratividade, principalmente em softwares de grande porte e relevância.

Em seu trabalho, SOARES (2004a) cita uma pesquisa realizada pela Standish Group. Segundo o site oficial da empresa, a Standish Group consiste em uma organização internacional e independente que realiza pesquisas na área de tecnologia da informação. Fundada em 1985, esta empresa é conhecida por seus relatórios explorando dados sobre projetos de software nos setores público e privado.

Em 1995, utilizando como amostra 8380 projetos de software, a Standish Group disponibilizou um conjunto de dados estatísticos sobre a indústria de desenvolvimento de software. Esses dados mostraram que um percentual muito pequeno de projetos de software era entregue de acordo com prazos, custos e requisitos estabelecidos, cerca de 16,2%. Outra estatística relevante publicada mostrava que mais de 30% dos projetos de software analisados foram cancelados. Os 52,7% de projetos que foram entregues apresentavam alterações nas definições iniciais realizadas, sejam elas de custo, prazo ou escopo.

Outro dado alarmante publicado foi a média de atraso e de aumento de custo percentual existente, 222% e 189% respectivamente. Além disso, considerando os 52,7% de projetos que apresentaram inconsistências com as definições iniciais quando entregues, verificou-se uma redução de quase 40% no escopo de funcionalidades.

Além de todos os dados negativos citados, sendo as características do modelo Clássico vistas como principais responsáveis por esses impactos, a pequena porcentagem de projetos entregues no prazo apresentava problemas de qualidade. Acredita-se que o motivo para a baixa qualidade dos produtos analisados seja a pressão exagerada comumente exercida sobre equipes de desenvolvimento de software, buscando maior assertividade de prazo nas entregas. A mesma pesquisa publicada pela Standish Group menciona que o número de erros apresentados por um

software pode até quadruplicar quando existem cobranças demasiadas sobre os responsáveis pelo desenvolvimento do projeto. Concluindo suas afirmações, SOARES (2004a) deixa claro que a indústria de software deveria adotar modelos incrementais de desenvolvimento, pois estes já se mostram possíveis soluções para os problemas apresentados.

2.2 METODOLOGIAS ÁGEIS DE DESENVOLVIMENTO DE SOFTWARE

Assim como qualquer indústria, a engenharia de software busca constantemente por oportunidades de melhoria em processos e lucratividade. Como o desenvolvimento de um software envolve diferentes atividades, as empresas estão sempre à procura de metodologias de desenvolvimento que definam processos capazes de resultarem em maior eficiência no uso de tempo e de recursos, e que possam gerar produtos de alta qualidade.

Consideradas uma grande inovação da indústria de software, as metodologias ágeis de desenvolvimento surgiram como uma possibilidade de criar produtos de maior qualidade e capazes de atender as expectativas dos clientes. O processo de desenvolvimento definido por essas metodologias permite que o software seja entregue no prazo correto e com alta eficiência no uso de recursos. O fluxo de atividades extremamente organizado definido pelas metodologias ágeis mostra-se efetivo (FADEL e SILVEIRA, 2010).

SOARES (2004a) comenta em seu trabalho que o termo "Metodologias ágeis" tornou-se popular a partir do ano 2001, quando foi realizada uma reunião entre dezessete especialistas focados nos processos de desenvolvimento de software. Entre esses especialistas estavam Schwaber e Beedle representando o modelo Scrum de desenvolvimento (SCHWABER e BEEDLE, 2002), Beck representando o modelo Extreme Programming (BECK, 1999), além de outros pesquisadores da área. Todos em conjunto estabeleceram características e princípios que eram compartilhados por essas metodologias de desenvolvimento emergentes, definindo e criando o chamado "Manifesto Ágil" e a Aliança Ágil (BECK et al., 2001). Apesar das várias definições existentes no "Manifesto Ágil", existem quatro princípios chave que são:

- Indivíduos e interações em vez de processos e ferramentas;
- Software executável em vez de documentação;
- Colaboração do cliente em vez de negociação de contratos;

- Respostas rápidas a mudanças em vez de seguir planos.

Analisando os princípios chave do "Manifesto Ágil", primeiramente se pode ter a ideia de que os conceitos utilizados nas metodologias tradicionais de desenvolvimento, como a importância dada a processos e ferramentas, a documentação, a negociação com clientes e até mesmo o planejamento do desenvolvimento de um software, para as metodologias ágeis seriam descartáveis.

As metodologias ágeis não rejeitam esses conceitos e atividades, apenas definem que os mesmos são secundários se comparados aos conceitos chave do "Manifesto Ágil". Considerando o contexto moderno da indústria de desenvolvimento de software, as metodologias ágeis se adaptam melhor às empresas de pequeno e médio porte, trazendo maior eficiência, lucratividade e capacidade de adaptação às mudanças. Scrum e XP (Extreme Programming) podem ser consideradas duas das metodologias ágeis mais conhecidas (SOARES, 2004a).

SOARES (2004a) cita que desde a primeira aplicação da metodologia XP de desenvolvimento de software a um projeto, excelentes resultados foram observados. O projeto C3 da Chrysler é considerado o primeiro a utilizar essa metodologia (ANDERSON, BEATTIE, BECK, 1998). A oportunidade surgiu devido a resultados ruins observados por vários anos com o uso das metodologias tradicionais de desenvolvimento no projeto. Ao aplicar a metodologia XP, o projeto foi concluído em pouco tempo e com o sucesso esperado. Observando o exemplo do projeto C3, fica claro que as metodologias ágeis de desenvolvimento de software surgiram de uma necessidade de modelos eficientes, já que os tradicionais já não se mostravam adequados.

Com o surgimento crescente de empresas de pequeno e médio porte na indústria de desenvolvimento de software, processos inflexíveis e caros de desenvolvimento como os definidos pelo modelo Cascata tornaram-se inviáveis. Além disso, a inflexibilidade defendida por esse modelo passou a cada vez mais se mostrar um fator limitante. Empresas pequenas, por não se adaptarem aos processos definidos pelos modelos tradicionais de desenvolvimento, muitas vezes não utilizavam processos formais nos seus modelos de desenvolvimento, e isso começou a gerar resultados desastrosos em relação à qualidade dos softwares produzidos. Todo esse contexto foi extremamente propício para o surgimento das metodologias ágeis (SOARES, 2004b).

Apesar do impacto positivo e das mudanças causadas no mercado, as metodologias ágeis de desenvolvimento não trouxeram nenhuma característica disruptiva consigo. A verdade é que, quando comparadas com as metodologias tradicionais, as principais diferen-

ças observadas estão no foco e nos princípios utilizados. Enquanto as metodologias tradicionais dão muito valor a processos e codificação, as ágeis valorizam mais as pessoas. Enquanto as metodologias tradicionais prezam pelas estimativas e preditibilidade, as ágeis focam na capacidade de adaptação às mudanças (SOARES, 2004b).

Novamente considerando o contexto moderno da engenharia de software e todas as revoluções tecnológicas que ocorreram nas últimas décadas, a capacidade de adaptação tornou-se fator essencial para o sucesso. Apesar de diferentes em muitos aspectos, principalmente em relação a algumas atividades adotadas, as metodologias ágeis compartilham muitos conceitos entre si. Em relação a esses conceitos compartilhados, podem ser citados o desenvolvimento iterativo e incremental, e o uso constante da comunicação superando documentação. Essas duas características são de extrema importância, pois se mostraram capazes de atender as expectativas de clientes cada vez mais críticos (SOARES, 2004b).

Para facilitar a visualização da importância do processo iterativo e adaptável de desenvolvimento defendido pelas metodologias ágeis, no gráfico apresentado pela Figura 2 temos a representação do aumento do custo de alterações de requisitos em cada fase do modelo Cascata. É possível perceber pela imagem que o aumento de custo pode ser exponencial. Uma mudança de requisito, quando realizada na fase de implantação de software do modelo Cascata, pode custar até 100 vezes o valor que seria gasto se a mesma fosse realizada durante a etapa de definição de requisitos. Levando isso em consideração, conclui-se que o modelo de desenvolvimento Cascata não é adaptável, pois pequenas mudanças nas expectativas sobre o produto final podem implicar em extrema ineficiência no uso de tempo e recursos.

O gráfico ideal para o aumento de custo oriundo de mudanças nos requisitos de um software é o exemplificado na Figura 3. Neste gráfico é possível perceber que o custo não aumenta exponencialmente mesmo nas fases finais do desenvolvimento de produto, fato importante já que mudanças em requisitos são muito comuns na indústria de software. A Figura 3 mostra o aumento de custo para alterações de requisitos quando um projeto está utilizando alguma das metodologias ágeis de desenvolvimento. É importante ressaltar que as metodologias ágeis incentivam mudanças constantes nos requisitos, até que o produto esteja adequado às expectativas do cliente (SOARES, 2004b).

A questão mais relevante para o contexto do presente trabalho em relação às metodologias ágeis de desenvolvimento de software, é a grande importância dada para os testes automatizados como con-

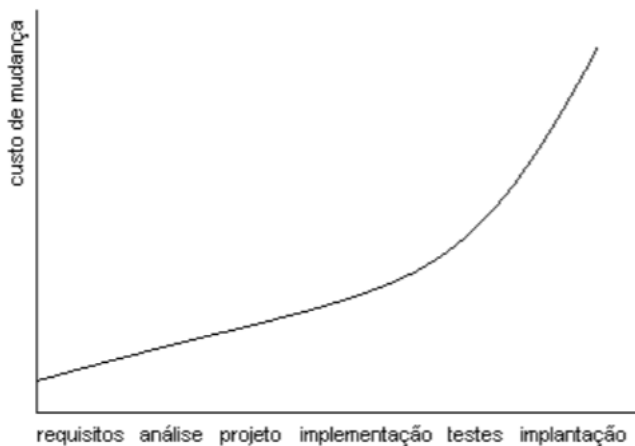


Figura 2 – Custo das mudanças de requisitos nas etapas do modelo cascata. Fonte: Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software

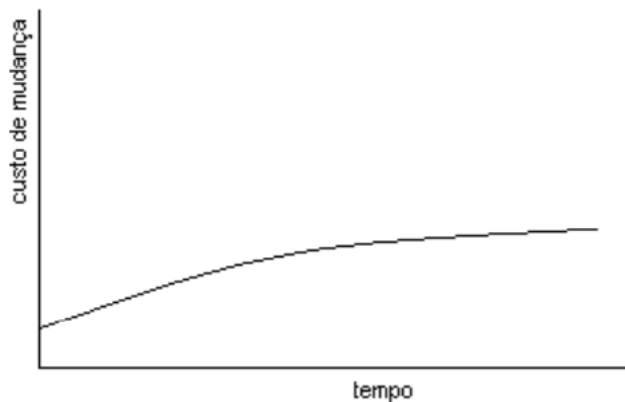


Figura 3 – Custo das mudanças de requisitos nos modelos ágeis de desenvolvimento. Fonte: Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software

sequência das constantes mudanças de requisitos e ao modelo incremental de desenvolvimento. Várias metodologias ágeis citam a importância dos testes no desenvolvimento de um software. A metodologia XP (Extreme Programming) por exemplo, define uma prática em que o desenvolvimento de um software deve ser baseado em testes (BECK, 1999).

2.3 QUALIDADE DE SOFTWARE

SILVA (2012) define com base nas terminologias de engenharia de software definidas pela IEEE - Institute of Electrical and Electronics Engineers - qualidade de software como a capacidade de um produto de atingir as expectativas dos clientes, independentemente de quais sejam. BARTIÉ (2002) define qualidade de software como sendo a conformidade de processos e produtos evitando defeitos. PRESSMAN (2005) a define como a conformidade às expectativas do cliente e a normas de desenvolvimento estabelecidas.

Sommerville (2011) define SQA - Garantia da Qualidade de Software - como um processo que visa estabelecer rotinas que possam não só garantir a qualidade de um software, mas também definir quais seriam as características que se adaptam às expectativas criadas sobre ele. A qualidade de um software não é definida apenas pelo próprio produto. Os processos utilizados para desenvolvimento de um sistema também são avaliados quanto a qualidade, pois influenciam diretamente nos resultados esperados pelos interessados em um projeto de software.

SILVA (2012) conclui que a garantia de qualidade de software é um processo cujos objetivos principais são validar que um produto e que todo o processo de desenvolvimento existente para a criação do mesmo estão de acordo com o esperado por todos os envolvidos no projeto. Relembrando novamente o conceito de custo para correções de erros em um software, e que o mesmo aumenta ao longo do ciclo de desenvolvimento do produto independentemente da metodologia utilizada, o processo de garantia de qualidade pode aumentar a eficiência e lucratividade de um projeto de software, detectando problemas nas fases iniciais de desenvolvimento.

2.3.1 Testes de software

DELAMARO et al. (2016) afirmam que o processo envolvido na execução de testes de software é complexo, pois existem muitas variáveis que podem implicar na ocorrência de erros e falhas em um software. Para exemplificar essa afirmação, os autores citam os erros funcionais que um software pode apresentar, que seriam a não conformidade com os requisitos de funcionalidade propostos, e os compara com falhas de segurança que um software pode conter. Apesar de serem dois tipos de erros diferentes e que se encontram em camadas distintas de um software, é dever dos testes de software encontrá-los a tempo de serem corrigidos sem consequências graves ao projeto.

Levando em consideração essa característica dos softwares, a atividade de testes precisou ser dividida em diferentes etapas e técnicas, capazes de validar todas as camadas de um produto. Os autores citam como principais fases o teste de unidade, o teste de integração, o teste de sistemas e o teste de regressão.

Os autores discorrem inicialmente sobre os testes de unidade, utilizados para garantir o funcionamento correto das menores frações de um software, que podem ser funções, procedimentos, métodos ou classes. É necessário garantir que cada pequena parte de um programa funcione corretamente para que o mesmo, em sua totalidade, seja confiável. Por meio dos testes de unidade são encontrados problemas de codificação, de uso incorreto de estrutura de dados, de lógica de algoritmos, entre outros.

Os testes de unidade permitiram a criação de metodologias que defendem que o desenvolvimento de um software deve ser baseado em testes, isto porque como é possível testar cada função separadamente, tornou-se viável utilizar esse tipo de teste no decorrer do desenvolvimento do software. Os outros tipos de teste por sua vez, necessitam do software totalmente finalizado para serem utilizados, fato que faz com que o desenvolvimento do produto deva ocorrer antes da execução de testes.

Um tipo de teste que obrigatoriamente só pode ser executado após unidades de um programa estarem prontas, é o teste de integração. Apesar de ainda ser um tipo de teste que busca validar as unidades de um software, diferentemente dos testes de unidade os testes de integração, como o próprio nome sugere, buscam avaliar o funcionamento das unidades quando integradas uma as outras. Devido a isso, podemos dizer que é um tipo de teste que valida a estrutura do sistema.

Garantir que uma unidade funciona sem erros isoladamente, não

garante que a mesma irá executar suas funções corretamente quando integrada a outras unidades. Por isso os testes de integração são de extrema importância. Como para realizar esse tipo de teste é necessário possuir um grande conhecimento sobre a estrutura interna de um software, geralmente esse tipo de teste, assim como os testes de unidades, também é executado pela própria equipe de desenvolvimento.

Durante o desenvolvimento de um software, é possível realizar testes de unidade e de integração para garantir que as partes já codificadas funcionam corretamente. Porém, quando se fala de testes de sistema, uma das premissas necessárias para execução desse tipo de teste é possuir o sistema já completo disponível para validação. É necessário que todas as partes do mesmo já estejam finalizadas e integradas.

Esse tipo de teste, por ser executado apenas ao final do processo de desenvolvimento, busca validar se o comportamento do software está de acordo com os requisitos definidos em conjunto com o cliente. Nesta fase de testes também são avaliados requisitos não funcionais de software, como nível de segurança do sistema, de performance, entre outros pontos. Para realizar a execução dos testes de sistema, geralmente equipes externas à equipe de desenvolvimento são designadas.

Os três tipos de testes citados são os mais conhecidos e utilizados na prática, porém existem muitos outros que também são aplicados nas empresas. Um tipo de teste extremamente relevante para o presente trabalho é o teste de regressão ou regressivo. Esse tipo de teste apresenta algumas diferenças em relação aos três tipos principais, começando pelo momento em que são utilizados, já que os testes regressivos geralmente são executados na etapa de manutenção do software. Isso porque, o objetivo principal de um teste de regressão é garantir que o software continua funcionando após novas funcionalidades ou alterações serem inseridas.

É comum que um software demonstre problemas em diferentes unidades após uma delas sofrer alterações, ou após uma nova unidade ser integrada ao sistema. Por isso os testes de regressão mostram-se indispensáveis. Quando pensamos nas metodologias ágeis, podemos ligar os testes de regressão com as entregas constantes e o fluxo iterativo de desenvolvimento. Em processos de desenvolvimento que utilizam metodologias ágeis, os testes regressivos são constantes e podem se tornar caros, isso porque geram retrabalho. Como consequência desse problema, os testes automatizados ganharam extrema importância no uso das metodologias ágeis.

DELAMARO et al. (2016) também destacam em seu trabalho que, independentemente do tipo de teste a ser utilizado, existem três

etapas bem definidas executadas durante a atividade de testes. São elas:

- Planejamento;
- Projeto de casos de teste;
- Execução e análise.

2.3.2 Testes automatizados de software

BERNARDO e KON (2008) afirmam em seu trabalho que, pensando nos modelos tradicionais de desenvolvimento e nos processos definidos por esses modelos, geralmente o desenvolvimento de um software ocorre seguindo atividades padronizadas. Inicialmente se realiza uma análise sobre o problema apresentado, posteriormente modelando uma solução de software e finalmente codificando a mesma. É comum que somente após essas três etapas concluídas, os próprios desenvolvedores realizem testes sobre o software implementado, geralmente de forma manual.

Quando um erro é encontrado durante a execução dos testes manuais, os desenvolvedores precisam corrigir o mesmo e executar todos os testes manuais novamente para garantir que o software como um todo atenda ao esperado. Muitas vezes ocorre também a execução de testes manuais por outros interessados no projeto, como clientes ou uma equipe externa de testes. Esse processo ocorre antes que um sistema seja colocado em produção.

Em organizações que utilizam metodologias de desenvolvimento rígidas e com processos bem definidos, muitas vezes a situação citada ocorre. Essas metodologias, sendo muitas delas derivadas dos modelos clássicos, geram muitos problemas comuns na indústria de software. Em relação aos oriundos da dificuldade de gerenciamento e execução de testes manuais, é possível citar o não cumprimento de prazos, a grande quantidade de bugs encontrados nos softwares codificados, e a enorme dificuldade em realizar a manutenção e evolução de produtos.

O problema não são os testes manuais em si, mas a repetição excessiva de um conjunto de testes muito extenso, que acaba por gerar retrabalho e desperdício de recursos. Apesar de a execução de um simples teste manual ser rápida e barata, é importante que esse tipo de teste seja utilizado com inteligência. Além dos custos envolvidos na execução repetitiva de um conjunto de testes manuais, é comum que os responsáveis pela execução não realizem-na por completo devido ao

processo cansativo e tedioso de executar os mesmos testes continuamente.

É um processo no qual é muito comum e compreensível ocorrerem falhas humanas, resultando em bugs no produto entregue ao cliente. E como os custos para identificar e corrigir um bug nesta etapa de desenvolvimento são muito elevados, conseqüentemente erros diminuem muito a eficiência do projeto, causando atrasos nas entregas e muitas vezes gerando softwares de qualidade duvidosa.

Como o esforço para executar toda a suíte de testes de regressão de forma manual é muito elevado, geralmente apenas uma parte dos testes é utilizada a cada nova alteração ou adição de código ao software. Por mais que as unidades de um sistema estejam bem mapeadas, é comum que alterações em uma certa unidade causem problemas em outras unidades pouco relacionadas. Com o passar do tempo, muitos erros são introduzidos ao software e a atividade de manutenção se torna mais complexa e cara, gerando um efeito chamado "bola de neve". Se o problema não for mitigado por meio de reestruturação de processos, é comum que em certo momento a manutenção do sistema seja tão ineficiente que produzir um novo software para substituí-lo se torne a melhor opção.

O processo de garantia de qualidade de software deve ser iterativo e constante. Prevenir defeitos, apesar de ser uma atividade que possui custos, se mostra mais eficiente do que identificar e corrigir bugs. As metodologias ágeis defendem que todos os envolvidos em um projeto de software devem possuir esse pensamento e buscar ao máximo garantir a qualidade do produto, sejam eles programadores, gerentes, testadores ou clientes.

As principais metodologias ágeis, como por exemplo Scrum e XP, compartilham dessa ideia. A metodologia XP vai além, e recomenda que testes automatizados sejam utilizados durante o processo de desenvolvimento de software, pois são efetivos para a garantia de qualidade. Além das recomendações presentes no discurso dos métodos ágeis de desenvolvimento, atividades adicionais que possam aumentar a qualidade dos produtos são bem aceitas.

Testes automatizados podem ser conceituados como rotinas que, com pouca ação humana envolvida podem simular várias funcionalidades presentes em um software, validando todas as respostas e possíveis erros gerados como resultados. Os testes automatizados geralmente demandam conhecimento de programação para serem implementados e utilizados. Como a interação humana é muito pequena, geralmente é possível executar uma suíte de testes automatizados muitas vezes

de forma fácil e barata, além de pouco sujeita a falha humana já que a maior parte da atividade é executada por computadores. Quando um teste manual é realizado, é comum que detalhes sejam esquecidos ou ignorados e que valores de entrada e saída sejam utilizados incorretamente. Em situações como essa, muitos bugs podem não ser encontrados. Uma das vantagens da automatização de testes é a menor ocorrência de falhas humanas.

Além de menos suscetíveis a falhas humanas, testes automatizados podem simular situações extremamente complexas por serem executados por computadores. É possível validar uma quantidade muito superior de combinações e contextos do que seria possível manualmente. Por exemplo, simular um grande número de usuários utilizando um software ou uma grande quantidade de operações ocorrendo simultaneamente, de forma manual é inviável e muito caro. Simular as mesmas situações de forma automatizada pode ser simples e rápido.

A automatização de testes se mostra uma solução efetiva para os problemas apresentados pelos testes manuais, principalmente no contexto das metodologias ágeis de desenvolvimento de software. Aplicar testes automatizados ao processo de desenvolvimento pode trazer muitos benefícios para uma empresa, principalmente diminuir a quantidade de bugs presentes nos softwares entregues a clientes, tornando o custo para a implementação de um sistema menor e a qualidade percebida maior. Como a execução de suítes de testes automatizados pode ser realizada constantemente, as equipes de desenvolvimento podem implementar novas funcionalidades com muito mais segurança.

Ao final, conclui-se que a utilização de testes automatizados pode aumentar a vida útil de um software quando bem estruturada.

2.3.3 Norma IEEE 829

CRESPO et al. (2004) citam em seu trabalho a Norma IEEE 829 (IEEE, 2008) como um conjunto de diretrizes a serem aplicadas na atividade de teste de software. Essa norma é composta por oito documentos que visam guiar as principais etapas do processo de teste: planejamento dos testes, especificação dos casos de teste, e análise dos resultados de testes. Os documentos definidos pela norma são os seguintes:

- Plano de teste: define o planejamento para a execução da atividade de testes e é composto por itens como escopo de execução, técnicas de teste utilizadas, recursos necessários e prazos. Este

documento também define as funcionalidades e os requisitos não funcionais a serem validados, as atividades obrigatórias durante os testes e os riscos envolvidos;

- Especificação de projeto de teste: baseado no plano de teste, porém com uma abordagem mais aprofundada e detalhista. Assim como o plano de teste, identifica os requisitos funcionais e não funcionais a serem validados, define os casos de teste, os métodos de teste e os critérios de aprovação;
- Especificação de caso de teste: é composto por um maior detalhamento dos casos de teste a serem utilizados, incluindo dados de entrada e saída, e resultados esperados;
- Especificação de procedimento de teste: visa detalhar o processo de execução do conjunto de casos de teste, informando os passos necessários para a execução;
- Diário de teste: visa detalhar cronologicamente os acontecimentos importantes relacionados com o projeto de testes em execução;
- Relatório de incidente de teste: documento onde qualquer acontecimento importante que ocorra durante o projeto de testes e demande análise futura deve ser detalhado;
- Relatório-resumo de teste: como o próprio nome sugere, nesse documento são apresentados de forma resumida os resultados do projeto de testes, incluindo análises sobre os resultados obtidos;
- Relatório de encaminhamento de item de teste: necessário apenas quando existe uma separação entre a equipe de desenvolvimento e a equipe responsável pelos testes, e identifica os itens encaminhados para validação.

De maneira geral, existem três etapas distintas definidas pela norma, onde cada etapa utiliza um conjunto de documentos diferente. Essas etapas são: preparação do teste, execução do teste e registro do teste. Apesar de todas as definições existentes na norma, na prática cada projeto deve adaptar o modelo de acordo com as suas necessidades.

Projetos de desenvolvimento de software de pequeno porte, geralmente utilizam apenas uma parte dos documentos propostos pela norma, buscando assim diminuir os custos envolvidos no gerenciamento do projeto. Tal situação ocorre no presente trabalho, onde por ser realizado por uma equipe pequena e possuir baixa complexidade, utilizou-se

apenas de uma parte dos documentos propostos pela norma, sendo eles: plano de teste, especificação de caso de teste e relatório-resumo de teste.

2.4 BEHAVIOR-DRIVEN DEVELOPMENT

Em conjunto com o surgimento das metodologias ágeis, práticas e métodos inovadores de desenvolvimento tornaram-se realidade na indústria de software. Desenvolvimento orientado ao comportamento, mais conhecido como BDD (Behavior-Driven Development), é um dos métodos de desenvolvimento de software que surgiu para tornar a adoção dos princípios ágeis mais acessíveis.

Como consequência do crescimento da utilização de metodologias ágeis pelas empresas de software, o BDD evoluiu e tornou-se capaz de ser aplicado não só no processo de codificação de software, como também nas etapas de análise e validação. Esse método de desenvolvimento utiliza especificação de funcionalidades por meio de exemplos, técnica conhecida como Specification by Example (MORAES, 2016).

Em seu trabalho, CHIAVEGATTO et al. (2013) afirmam que uma das principais características do BDD é promover a comunicação entre todos os interessados no projeto de software, com um detalhe muito importante, utilizando uma linguagem compreensível por todas as partes envolvidas. Desta forma, todos podem contribuir para um melhor entendimento do problema que precisa de resolução, facilitando a definição e atualização constante dos requisitos do software.

A técnica se adapta muito bem a necessidade de feedback constante definida pelas metodologias ágeis. Por meio do BDD, histórias de usuário são especificadas e tendem a se transformar em funcionalidades do software. BDD é considerado uma evolução do TDD (Test Driven Development), porém enquanto no TDD o elemento de destaque são os testes de software, no BDD o item de maior relevância é o comportamento do software.

Por meio de uma comunicação efetiva e que permite a compreensão de todos os stakeholders relacionados a um projeto de software, o BDD busca diminuir ao máximo a ocorrência de falhas de comunicação que muitas vezes ocorrem em decorrência do uso de termos técnicos e inadequados ao entendimento de todos os envolvidos. A linguagem utilizada é comum e simples, baseada em termos de uso cotidiano. Para que o BDD possa ser aplicado adaptando-se a realidade das empresas, muitas ferramentas de suporte foram criadas. Existem ferramentas de apoio ao BDD em várias linguagens de programação diferentes, possi-

bilitando o uso das mesmas por diferentes públicos (CHIAVEGATTO et al., 2013).

Observando algumas práticas de engenharia de software, é fácil perceber que o BDD se resume a adoção de um conjunto delas. E assim como muitas das práticas de engenharia de software existentes, o BDD objetiva possibilitar o desenvolvimento de produtos de maior qualidade, respeitando sempre prazos e escopos definidos. Por meio da colaboração entre os stakeholders de um projeto, o BDD visa definir requisitos por meio de exemplos.

O BDD contém elementos de automação, de testes de software e de critérios de aceite, porém não visa a criação de nenhum desses itens. Essa prática resulta da aplicação em conjunto de uma linguagem acessível, do TDD (Test-Driven Development) e de testes de aceitação automatizados, extraindo o melhor possível de cada uma dessas técnicas (MORAES, 2016).

Permitir que todos os envolvidos no projeto de um software compartilhem informações dos mais variados tipos e origens, sejam elas técnicas ou de negócio, além de automaticamente refletir mudanças realizadas no projeto em todos os seus aspectos, desde requisitos ao código, são as maiores vantagens de se utilizar BDD. A integração entre os componentes e stakeholders do projeto é fortalecida e facilitada.

Por meio da ligação existente entre código e requisitos, torna-se mais fácil o gerenciamento da evolução do produto, e isso é válido para todo o ciclo de vida do software. Para verificar essas vantagens na prática, é necessário analisar a técnica Specification by example (SBE), considerada uma característica chave do BDD. Quando o comportamento esperado por um software é descrito na forma de exemplos e conversas, todos os envolvidos tendem a compreender de forma semelhante o que deve ser feito (MORAES, 2016).

MORAES (2016) complementa o seu trabalho definindo especificações executáveis como um dos resultados esperados ao se utilizar BDD. Tais especificações são úteis como base para a codificação de um software construído com uso do BDD. Por meio delas, os requisitos definidos pelos stakeholders são escritos com linguagens de programação. Concluindo, os princípios do BDD são definidos como sendo os seguintes:

- Foco no comportamento do sistema, e não nas suas especificações;
- Identificar qual comportamento entrega mais valor aos interessados;
- Utilizar exemplos para facilitar a compreensão do comportamento

```

Dado que um usuário deseja executar uma ação
E esse usuário está logado
Mas esse usuário não possui permissão para executar a ação
Quando o usuário tenta executar a ação
Então uma mensagem de erro é exibida

```

Figura 4 – Notação Gherkin. Fonte: Elaboração própria

desejado.

2.4.1 Gherkin

Gherkin consiste em uma notação utilizada para descrever comportamentos esperados por um software. Os exemplos utilizados no modelo BDD são escritos utilizando Gherkin. Na Figura 4 é possível observar um exemplo de cenário escrito utilizando essa notação. Existem várias ferramentas disponíveis para automatização de testes que possuem suporte a Gherkin. Por meio delas é possível transformar os cenários que descrevem o comportamento esperado por um software em uma suíte de testes automatizados.

Espera-se que funcionalidades complexas de um software possuam comportamentos diversos dependendo do contexto e das variáveis existentes. Os exemplos utilizados para descrever os comportamentos de uma mesma funcionalidade e escritos com uso da notação Gherkin, são agrupados para facilitar a compreensão dos interessados. O agrupamento é realizado utilizando arquivos de texto chamados arquivos de funcionalidade (feature files). Cada arquivo de funcionalidade possui exemplos que descrevem os comportamentos esperados para a mesma, além de uma descrição da funcionalidade (MORAES, 2016).

Segundo ARAUJO (2017), Gherkin auxilia no processo de criação de documentações automatizáveis baseadas nos comportamentos esperados por um sistema, conseqüentemente sendo útil para a automatização de testes. Essa notação consiste em uma linguagem objetiva, simples e estruturada, que facilita o entendimento dos interessados na documentação do software.

Apesar de documentações escritas com uso da notação Gherkin serem passivas de automatização, não é necessário descrever como os comportamentos do software serão implementados. É muito comum observar o uso da notação Gherkin para descrever cenários e exemplos de comportamento no modelo de desenvolvimento BDD. Por meio de

palavras chave (Given-When-Then) é possível tornar a descrição dos comportamentos esperados em testes de aceitação automatizados, além de facilitar o entendimento dos requisitos pelos envolvidos no processo.

2.5 API REST

Segundo RODRIGUEZ (2008), a arquitetura REST, modelo utilizado na estruturação de serviços Web, consiste em um conjunto de princípios e conceitos que guiam os desenvolvedores durante a construção de um software para a Web. Essa arquitetura é orientada aos recursos presentes no sistema em construção, definindo seus possíveis estados, como devem ser endereçados, e como devem ser compartilhados por meio do protocolo HTTP (Hypertext Transfer Protocol).

Os recursos de um software podem ser transferidos entre uma grande variedade de clientes por meio da arquitetura definida. O modelo REST é considerado o predominante entre os modelos existentes para construção de serviços Web. Isso é observado por meio da quantidade de softwares desenvolvidos com uso dessa arquitetura. Por ser um modelo mais simples e eficiente, o REST substituiu os modelos SOAP e WSDL, que eram os mais utilizados antes do seu surgimento. Um serviço Web construído com base na arquitetura REST deve seguir quatro princípios:

- Uso de métodos HTTP;
- Uso de requisições independentes (stateless);
- Uso de URI's baseadas na estrutura do projeto;
- Uso de XML, JavaScript Object Notation (JSON), ou ambos para transferência de dados.

2.5.1 Métodos HTTP

RODRIGUEZ (2008) cita em seu trabalho que o uso de métodos HTTP de maneira explícita é um dos princípios da arquitetura REST. É preciso observar que o protocolo citado pela arquitetura REST é o HTTP definido pela RFC 2616. Este documento técnico define uma série de características do protocolo HTTP. É possível citar por exemplo, a definição de que o método HTTP GET deve ser utilizado para recuperar recursos do serviço quando solicitados por um cliente.

Como a arquitetura REST orienta que os desenvolvedores utilizem métodos HTTP explicitamente e de maneira consistente com a definição do protocolo, uma estrutura com mapeamentos individuais é concebida entre as operações de criação, leitura, atualização e exclusão, e os métodos HTTP. De acordo com esses mapeamentos individuais, conclui-se que:

- Para criar um recurso utilizando um serviço Web, utiliza-se o método HTTP POST;
- Para recuperar um recurso utilizando um serviço Web, utiliza-se o método HTTP GET;
- Para atualizar um recurso utilizando um serviço Web, utiliza-se o método HTTP PUT;
- Para excluir um recurso utilizando um serviço Web, utiliza-se o método HTTP DELETE.

2.5.2 Stateless

RODRIGUEZ (2008) comenta que em um mercado cada vez mais competitivo e exigente, a arquitetura REST, devido aos princípios que define, tornou-se um excelente método para a construção de serviços Web de alto desempenho. Devido principalmente a sua característica de lidar com requisições de forma independente (stateless), em serviços Web estruturados com REST é possível utilizar estratégias de balanceamento de carga, tolerância a falhas (failover) e escalonamento de recursos, utilizando clusters de servidores para atender as requisições recebidas. Para diminuir o tempo de resposta de um serviço Web, são utilizados proxies e gateways que encaminham as requisições entre os servidores que compõe a topologia do serviço requisitado.

É importante destacar que o escalonamento de um serviço Web em diversos servidores só é possível devido ao princípio stateless, onde os clientes do serviço encaminham requisições de recursos completas e com todos os dados necessários. Se as requisições não possuíssem essa característica, não seria possível encaminhá-las entre os servidores da topologia do serviço, pois dados de estado armazenados localmente por cada servidor seriam perdidos a cada nova requisição encaminhada.

Como uma requisição completa não necessita que os servidores recuperem dados, contextos ou estados, isso porque as requisições recebidas já possuem no cabeçalho HTTP todas as informações necessárias

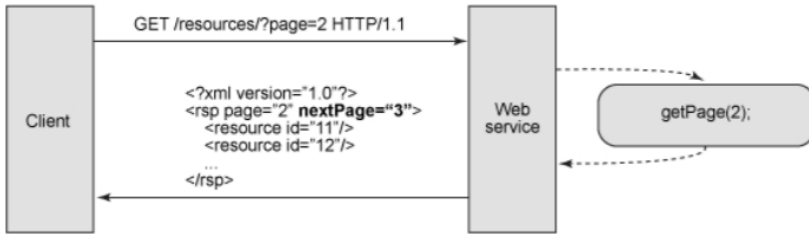


Figura 5 – Exemplo de comunicação cliente-servidor stateless. Fonte: RESTful Web services: The basics

para que o servidor possa gerar uma resposta, foi possível simplificar a arquitetura e melhorar a eficiência de serviços Web. A Figura 5 exibe um exemplo de comunicação cliente-servidor stateless.

2.5.3 URI

RODRIGUEZ (2008) cita o uso de URIs definidos com base na estrutura de diretórios do serviço Web como um dos princípios da arquitetura REST. Utilizar URIs baseados nessa estrutura é um dos métodos para atingir o objetivo esperado com uso das mesmas, que é indicar de forma intuitiva e simples como solicitar os recursos de um serviço Web. Os URIs devem servir como uma documentação desses recursos, de maneira que seja simples para um desenvolvedor entender o que pode ser acessado por meio dos mesmos.

As principais características de um URI devem ser a objetividade, a previsibilidade e a facilidade de compreensão. Quando um URI é definido com base na estrutura de diretórios do serviço Web, uma cadeia hierárquica de recursos é concebida, onde a partir de um caminho inicial ou raiz, os caminhos derivados definem as principais funções do serviço.

Algumas recomendações existentes para o uso de URIs em serviços Web baseados em REST são:

- Não devem ser exibidas as tecnologias utilizadas na implementação dos scripts existentes no servidor, para que seja possível modificar tais tecnologias quando necessário sem alterar os URIs;
- Os URIs devem ser escritos com uso de apenas letras minúsculas;
- Não devem ser utilizados espaços na composição dos URIs. Usam-se hifens ou sublinhados, nunca misturando os dois no mesmo

URI;

- Sempre que possível evitar o uso de parâmetros informados na URL (query strings);
- Para requisições enviadas a caminhos que não apontam para nenhum recurso do serviço Web, responder com um erro padrão e informativo.

Outra recomendação importante para o uso de URIs, é que por mais que um serviço Web possa ser dinâmico e alterado constantemente, os URIs utilizados para solicitar os recursos devem ser estáticos. Dessa maneira, mesmo quando a implementação de um serviço Web é alterada, o link para solicitar o recurso do serviço continua o mesmo. Como consequência, outra recomendação é que os relacionamentos entre os recursos do serviço Web apontados nos URIs sejam independentes dos relacionamentos existentes entre os mesmos recursos a nível de implementação.

2.5.4 Envio de dados em formato XML, JSON, ou ambos

Como último princípio da arquitetura REST, em seu trabalho RODRIGUEZ (2008) comenta sobre os formatos de dados utilizados. Seguindo com a proposta de simplicidade e eficiência, a arquitetura REST promove o uso de formatos de dados de fácil compreensão humana durante a comunicação cliente-servidor. As informações são transferidas entre os clientes e os serviços Web por meio do corpo das requisições HTTP.

Os relacionamentos comumente existentes entre os recursos de um serviço Web precisam ser refletidos durante a transferência de dados. A arquitetura REST utiliza os formatos de dados JSON, XML e XHTML para transferência de dados nas requisições HTTP, pois os mesmos possibilitam a obtenção de simplicidade e eficiência, representando de maneira eficaz e passível de compreensão humana os relacionamentos existentes entre os recursos de um serviço Web.

A transferência de dados, ocorrendo com uso de formatos padrões, permite que um serviço Web se comunique com clientes implementados em diferentes tecnologias, e executando em diferentes plataformas e dispositivos. No cabeçalho das requisições HTTP, um cliente pode informar qual formato de dado deseja utilizar durante a comunicação com o servidor. Essa possibilidade minimiza o acoplamento de dados entre o serviço Web e os clientes que desejam utilizá-lo. Na


```
<student id="1">
  <name>Rodrigo</name>
  <age>24</age>
  <institution>Universidade Federal de Santa Catarina</institution>
</student>
```

Figura 6 – Formato XML. Fonte: Elaboração própria

```
{
  "name": "Rodrigo",
  "age": "24",
  "institution": "Universidade Federal de Santa Catarina"
}
```

Figura 7 – Formato JSON. Fonte: Elaboração própria

Figura 6 é exibido um exemplo de formato de dado XML. Na Figura 7 por sua vez, é exibido um exemplo de formato de dado JSON.

2.6 HAWA

O Hawa consiste em um software desenvolvido pela equipe do LabSEC, laboratório pertencente à UFSC, cuja finalidade é a emissão de certificados digitais a usuários finais. O mesmo é utilizado pelas entidades de hierarquia mais baixa na cadeia de entidades certificadoras do ICP - Brasil. Segundo o site oficial do ICP - Brasil:

A Infraestrutura de Chaves Públicas Brasileira – ICP-Brasil é uma cadeia hierárquica de confiança que viabiliza a emissão de certificados digitais para identificação virtual do cidadão. Observa-se que o modelo adotado pelo Brasil foi o de certificação com raiz única, sendo que o ITI, além de desempenhar o papel de Autoridade Certificadora Raiz – AC-Raiz, também tem o papel de credenciar e descredenciar os demais participantes da cadeia, supervisionar e fazer auditoria dos processos.

A Figura 8 ilustra a hierarquia da ICP-Brasil. Segundo o site Benefícios e Aplicações da Certificação Digital:

A cadeia de confiança de certificação digital é a hie-



Figura 8 – Hierarquia ICP-Brasil. Fonte: site Benefícios e Aplicações da Certificação Digital

rarquia existente entre os componentes da ICP-Brasil. Estes componentes são a AC Raiz (Autoridade Certificadora Raiz), as ACs (Autoridades Certificadoras) de primeiro nível e segundo nível, as ARs (Autoridades de Registros), e, finalmente, o usuário final.

A utilização de recursos envolvendo certificação digital cresce de maneira acelerada no Brasil. Dados do ITI mostram que já existem cerca de 9 milhões de certificados digitais ativos no país, sendo que em 2019 foram emitidos aproximadamente 5 milhões (ITI, 2019).

A oportunidade para elaboração do presente trabalho surgiu da necessidade de garantir de forma automatizada o correto funcionamento da API REST pública de comunicação existente no software Hawa. Por meio deste, objetiva-se garantir o funcionamento correto dos endpoints da API passíveis de uso de testes automatizados de integração.

3 PLANEJAMENTO

Este capítulo apresenta todas as análises e definições realizadas previamente à implementação dos testes automatizados. São detalhados os processos de elaboração do plano de teste e dos casos de teste, e a definição de tecnologias e ferramentas a serem utilizadas.

3.1 PLANO DE TESTE

Foi elaborado um plano de teste para guiar a implementação dos testes automatizados de integração. O mesmo foi adaptado ao contexto do presente trabalho. É possível visualizar o documento na íntegra no Apêndice A.

Os objetivos do plano de teste são: identificar informações existentes e os componentes de software que devem ser testados; listar os requisitos de teste; descrever a estratégia de teste; identificar os recursos necessários; e listar os produtos de trabalho das tarefas de teste.

O escopo do trabalho foi definido como a implementação de testes automatizados de integração capazes de validar todos os endpoints da API REST do Hawa passíveis de validação automatizada.

Foram definidos como requisitos de teste: validar todos os endpoints de forma automatizada; implementar testes reutilizáveis; elaborar documento com instruções para uso dos testes; elaborar documento com instruções para uso dos testes em pipelines de validação.

Na tabela 1 é possível visualizar os recursos necessários para a execução do presente trabalho. Na tabela 2 é possível visualizar os produtos esperados como resultados do presente trabalho. Na tabela 3 estão descritas as atividades que devem ser realizadas durante a execução do presente trabalho.

Recurso	Descrição
Engenheiro de testes	Realizar planejamento, análise e implementação dos testes automatizados.
Ambiente de testes	Ambiente acessível e com o Hawa em execução.
Documentação	Documentação da API REST do Hawa.

Tabela 1 – Recursos

Produtos de Trabalho	Responsável
Plano de Teste	Rodrigo Bittencourt de Lima
Ambiente de Teste	Equipe de desenvolvimento do Hawa
Documentação de API REST	Equipe de desenvolvimento do Hawa
Casos de teste	Rodrigo Bittencourt de Lima
Scripts de Teste Automatizados	Rodrigo Bittencourt de Lima
Resultados de Testes	Rodrigo Bittencourt de Lima

Tabela 2 – Produtos de trabalho

Tarefa Macro	Tarefas
Planejar Teste	Identificar Requisitos para o Teste
	Desenvolver Estratégia de Teste
	Identificar Recursos de Teste
	Gerar Plano de Teste
Projetar Teste	Identificar e Descrever Casos de Teste
	Identificar e Estruturar Scripts de Teste
Implementar Teste	Configurar Ambiente de Teste
	Registrar ou Programar Scripts de Teste
Executar Teste	Executar Script de Teste
	Avaliar Execução do Teste
	Verificar os Resultados

Tabela 3 – Tarefas

3.2 ESPECIFICAÇÃO DOS CASOS DE TESTE

Foi elaborado um documento de Especificação dos casos de teste para guiar a implementação dos testes automatizados. Tal documento foi desenvolvido com base na documentação da API REST do software Hawa. Os casos de teste foram adaptados ao contexto e as necessidades do presente trabalho. É possível visualizar o documento na íntegra no Apêndice B.

Após validar em conjunto com a equipe responsável pelo Hawa quais eram os endpoints da API REST passíveis de validação automatizada, foram elaborados 17 casos de teste no total. Os casos de teste planejados são capazes de validar 5 endpoints do Hawa.

3.3 DEFINIÇÃO DE TECNOLOGIAS E FERRAMENTAS

Para a implementação dos testes automatizados de integração sobre API REST do software Hawa, objetivo principal do presente trabalho, se faz necessário o uso de alguma ferramenta capaz de executar tal tipo de teste. Foram analisadas, levando em consideração a utilização atual no mercado e experiências prévias dos envolvidos no trabalho, cinco ferramentas, sendo elas: Apache JMeter, Katalon Studio, SoapUI, Rest-Assured e Postman.

Após analisar as 5 ferramentas, optou-se por uma delas com base nos critérios: qualidade da documentação; custo; adoção da ferramenta pela indústria de software; possibilidade de implementação de testes automatizados; suporte a testes de API REST; facilidade de implementação de testes automatizados; possibilidade de uso de variáveis globais.

3.3.1 Apache JMeter

O Apache JMeter (Apache Software Foundation, 2019) consiste em um software de código aberto, originalmente implementado para ser utilizado em testes de sistemas Web. Utilizando tecnologias Java, acabou sendo expandido e tornando-se capaz de executar diferentes tipos de teste. A ferramenta pode ser utilizada para a execução de testes funcionais, testes de desempenho, testes de carga, entre outros. Como alguns dos recursos do Apache JMeter se pode citar:

- Testes de desempenho de diferentes tipos de sistema;

- Testes sobre serviços da Web projetos com arquiteturas SOAP ou REST;
- IDE (Integrated Development Environment ou Ambiente de Desenvolvimento Integrado) completo para suporte a implementação de testes;
- Utilizável em qualquer sistema operacional compatível com Java;
- Geração de relatórios em HTML com os resultados das execuções de testes;
- Capacidade de lidar com variados formatos de resposta (HTML, JSON, XML ou qualquer formato de texto);
- Entradas dinâmicas para testes e manipulação de dados.
- Integração contínua através de bibliotecas de código aberto de terceiros.

3.3.2 Katalon Studio

O Katalon Studio (Katalon LLC, 2019) atualmente é utilizado em 160 países ao redor do mundo, e consiste em uma ferramenta gratuita para automação de testes de API, Web e dispositivos móveis. É possível estender as funcionalidades presentes na ferramenta por meio de plugins disponíveis.

Possui uma interface gráfica completa e fácil de usar, permitindo que usuários com poucos conhecimentos em programação possam utilizar a ferramenta. Oferece também recursos avançados para usuários com maior conhecimento técnico.

3.3.3 SoapUI

SoapUI (SmartBear Software, 2019) é a ferramenta líder no mercado de testes automatizados de API. Devido a união de uma interface gráfica fácil de usar e recursos avançados, é a ferramenta mais utilizada para testes automatizados sobre API SOAP e REST. É possível implementar e executar testes funcionais, regressivos e de carga.

A ferramenta possui suporte para desde os testes mais comuns de serviços Web, até testes sobre camadas de mensagens. Foi a primeira ferramenta de código aberto para testes de API lançada, em 2006.

Fruto da colaboração de vários desenvolvedores, surgiu com o intuito de ajudar os profissionais envolvidos com a área de engenharia de software, e tornou-se a ferramenta de código aberto para testes de API SOAP e REST mais utilizada nos últimos dez anos.

As principais funcionalidade dessa ferramenta são:

- Testes funcionais e de regressão automatizados, para os quais, devido a facilidade da ferramenta, não é necessário ser um desenvolvedor para criá-los;
- Testes sobre serviços da Web projetados com arquiteturas SOAP ou REST;
- Não é necessário codificar os testes, pode-se realizar a criação dos mais variados cenários de teste, simples ou complexos, por meio da ações de arrastar e soltar;
- Construção de simuladores de serviços Web ainda não implementados, conhecidos como mocks;
- Testes de segurança;
- Testes de carga;
- Testes de carga baseados em testes funcionais;
- Validações de performance;
- Suporte a testes sobre os mais variados protocolos;
- Suporte a ferramentas de integração contínua;
- Por ser uma ferramenta de código aberto, permite a construção de plugins para expandir as suas funcionalidades.

3.3.4 Rest-Assured

Rest-Assured (HALEBY, 2010) é uma biblioteca utilizada para criação de testes automatizados, que objetiva trazer para as tecnologias Java a mesma simplicidade existente na validação de serviços Web REST construídos com linguagens dinâmicas como Ruby e Groovy. O código exemplo presente na Figura 9 exemplifica a simplicidade da ferramenta. A biblioteca possui funcionalidades importantes para a codificação dos testes automatizados, como:


```

@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
        get("/lotto/{id}", 5).
    then().
        statusCode(200).
        body("lotto.lottoId", equalTo(5),
            "lotto.winners.winnerId", hasItems(23, 54));

}

```

Figura 9 – Exemplo de código de teste implementado com Rest-Assured. Fonte: site oficial da ferramenta

- Vários mecanismos para simular autenticação de usuários;
- Suporte a qualquer método HTTP, incluindo a utilização de parâmetros, cabeçalhos e corpo de requisições.

3.3.5 Postman

O Postman (Postman Inc., 2019) consiste em uma ferramenta de apoio ao desenvolvimento de APIs. Os recursos que o Postman oferece podem ser utilizados para simplificar os processos envolvidos durante o desenvolvimento de uma API, sendo uma ferramenta excelente para fomentar a colaboração entre equipes de desenvolvimento durante a implementação de novos softwares. Por meio do Postman, é possível criar scripts automatizados de teste, unindo posteriormente um conjunto de scripts para compor uma suíte completa de testes automatizados de validação de API. Após a criação de uma suíte de testes, é possível executá-la sempre que for necessário validar o funcionamento correto de uma API. Utilizando essa ferramenta, é possível implementar e executar diversos tipos de testes, como por exemplo:

- Testes unitários;
- Testes funcionais;
- Testes de integração;

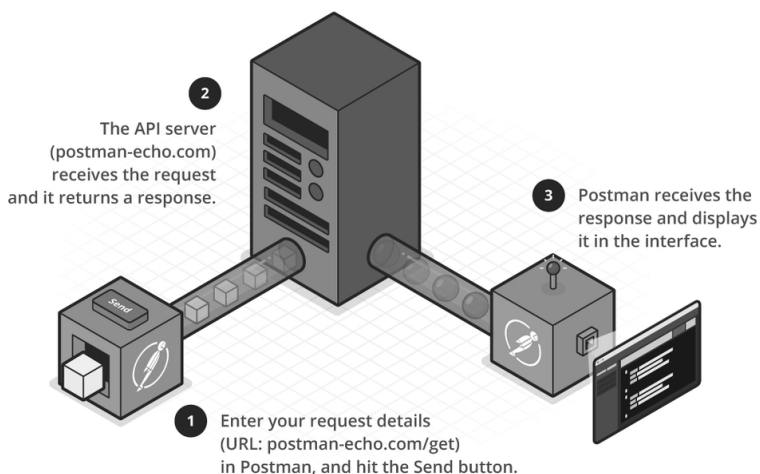


Figura 10 – Funcionamento do Postman. Fonte: documentação oficial da ferramenta

- Testes de ponta a ponta (end to end ou e2e);
- Testes de regressão;
- Testes simulados (testes com mocks).

A Figura 10 ilustra o funcionamento básico do Postman. O Postman utiliza o conceito de coleções para realizar o agrupamento de requisições configuradas. Dessa forma, torna-se simples o gerenciamento de múltiplas requisições, como também a configuração de suítes de testes automatizados.

A navegação pela ferramenta é simples e rápida devido a interface com o usuário bem organizada. A Figura 11 ilustra a interface básica da ferramenta. Além disso, existem muitos atalhos de teclado que facilitam ainda mais a utilização da ferramenta.

3.3.6 Definição de ferramentas

Após testes realizados com cada uma das 5 ferramentas citadas anteriormente, verificou-se que todas são poderosas o suficiente para a

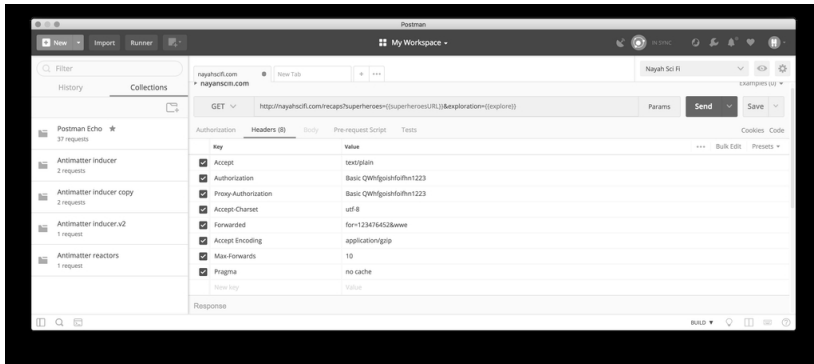


Figura 11 – Layout do Postman. Fonte: documentação oficial da ferramenta

execução com eficácia do trabalho proposto. Tal fato ampliou a gama de possíveis ferramentas a se utilizar para implementação dos testes automatizados.

Como já citado, os critérios para escolha da ferramenta definidos foram: qualidade da documentação; custo; adoção da ferramenta pela indústria de software; possibilidade de implementação de testes automatizados; suporte a testes de API REST; facilidade de implementação de testes automatizados; possibilidade de uso de variáveis globais.

Levando em consideração os critérios citados, decidiu-se por utilizar o Postman como ferramenta de implementação e execução dos testes automatizados sobre API REST, testes estes que são esperados como principal resultado do presente trabalho. O Postman mostrou-se uma ferramenta confiável e eficiente. Como citado no site oficial da ferramenta, a mesma já foi utilizada por mais de oito milhões de desenvolvedores, de mais de quatrocentas mil empresas em todo o mundo (Postman Inc., 2019).

4 IMPLEMENTAÇÃO

Este capítulo apresenta o processo de implementação dos testes automatizados. São apresentados o ambiente de testes utilizado e o processo de implementação dos testes automatizados utilizando a ferramenta Postman.

4.1 AMBIENTE DE TESTES

Para que seja possível a implementação e a execução dos testes automatizados sobre API Rest do software Hawa, faz-se necessário a disponibilização de um ambiente de testes. Este ambiente deve possuir uma instância do software Hawa em execução e acessível remotamente.

Devido a essa necessidade, a equipe responsável pelo software Hawa configurou e disponibilizou uma versão do mesmo em execução em servidor com acesso remoto liberado. Dessa forma, foi possível implementar e executar diversas vezes os testes automatizados, realizando ajustes até que os mesmos se encontrassem na forma ideal.

Além disso, verificou-se que, para futuras execuções e validações com os testes automatizados disponibilizados para a equipe responsável pelo software ao final do trabalho, basta configurar novamente um servidor com acesso remoto e com o software em execução, executando os testes por meio de uma instalação remota do Postman.

4.2 TESTES AUTOMATIZADOS COM POSTMAN

Todo o processo de implementação dos testes automatizados foi realizado com base na documentação oficial da ferramenta Postman. A mesma é suficientemente completa, objetiva e eficaz. Todas as informações a seguir são oriundas da documentação citada.

Os testes automatizados foram implementados com uso de sistema operacional Windows 10. A ferramenta Postman possui uma aplicação nativa para esse sistema, fato que faz com que o processo de instalação da ferramenta se resuma à execução do instalador encontrado facilmente no site da mesma. Opcionalmente é possível configurar uma conta de usuário inicialmente gratuita.

A funcionalidade do Postman que possibilitou a utilização do mesmo no presente trabalho foi a capacidade de execução de scripts

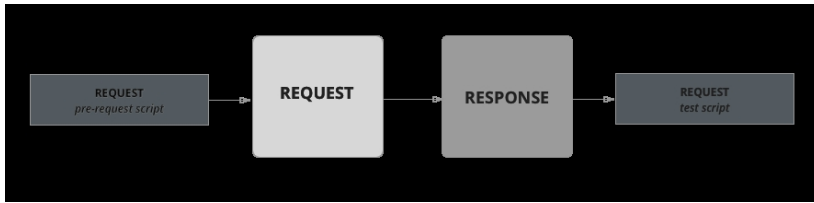


Figura 12 – Adição de scripts à requisições no Postman. Fonte: documentação oficial da ferramenta

por meio de uma poderosa ferramenta baseada em Node.js (Node.js Foundation, 2019). Essa ferramenta é capaz de executar scripts escritos com a linguagem de programação JavaScript, e foi construída sobre o motor de execução Chrome's V8. Devido a essa ferramenta, é possível implementar com o Postman suítes de testes automatizados e utilizar parâmetros dinâmicos.

Ao configurar uma requisição no Postman, é possível adicionar scripts JavaScript para execução antes do envio da requisição, como também após o recebimento da resposta da mesma. Os testes automatizados são executados após o recebimento da resposta de uma requisição enviada com o Postman. A Figura 12 ilustra essa dinâmica.

Para cada caso de teste desenhado na etapa de planejamento do presente trabalho, foi configurada uma requisição no Postman. Para cada requisição configurada, um script de teste foi codificado para validar a resposta recebida ao enviar a requisição. No Apêndice C estão descritas todas as requisições configuradas, omitindo alguns detalhes devido a privacidade do software em teste.

Foram implementados casos de teste com parâmetros válidos e também com parâmetros inválidos para validar o funcionamento do software em ambas as situações. Espera-se que o software se comporte corretamente ao receber parâmetros válidos, e que seja capaz de fornecer feedbacks adequados ao receber parâmetros inválidos.

Com intuito de exemplificar o processo de validação automatizada utilizado, optou-se por descrever o caso de teste nomeado "Criar certificate application com sucesso". Esse caso de teste busca validar que a API REST do Hawa comporta-se corretamente ao receber uma requisição válida solicitando a criação de um pedido de identificação virtual de cidadão.

Inicialmente, configurou-se uma requisição para o endpoint `"/create"` utilizando o parâmetro `"profile"` contendo um valor válido dentro do

```

pm.test('returns a 200 response', () => {
  pm.response.to.have.status(200);
});

pm.test('returns the protocol number of the created application', () => {
  let jsonData = pm.response.json();
  pm.expect(jsonData.protocol).to.be.at.least(1);
});

pm.test('returns the correct registration authority response', () => {
  let jsonData = pm.response.json();
  pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("201 Created");
});

pm.test('returns the correct error message', () => {
  let jsonData = pm.response.json();
  pm.expect(jsonData.errorMessage).to.eql("no error");
});

let jsonData = pm.response.json();
pm.environment.set("protocol", jsonData.protocol);

```

Figura 13 – Script de teste. Fonte: Elaboração própria

esperado pela API. Como resposta ao envio da requisição esperava-se que o retorno da API possuísse um status HTTP positivo (HTTP 200) e um JSON contendo os seguintes valores: o dado chamado "protocol" contendo um número inteiro positivo; o dado "registrationAuthorityResponse.status" contendo o valor "201 Created"; e o dado "errorMessage" contendo o valor "no error".

Para validar o retorno da API de forma automatizada, foi codificado o script de testes exibido na Figura 13. A requisição configurada é exibida na Figura 14.

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	profile		

Figura 14 – Requisição de teste. Fonte: Elaboração própria

5 ANÁLISE DE RESULTADOS

Este capítulo apresenta os resultados obtidos com a implementação e execução dos testes automatizados de integração sobre o software Hawa.

5.1 RESULTADOS POSITIVOS

Após a automatização dos 17 casos de teste, totalizando cerca de 60 itens validados, sendo essa implementação realizada de forma colaborativa em conjunto com a equipe responsável pelo Hawa para que as validações estivessem de acordo com o esperado, a maioria dos resultados foram positivos.

Executando a suíte de casos de teste ao final da implementação, foi possível concluir que os endpoints da API REST pública do Hawa se comportam conforme o esperado para a maioria das validações, tanto em testes utilizando dados válidos como em contextos envolvendo dados inválidos. Isso vale para todos os endpoints passivos de automatização. De 63 validações realizadas, 57 tiveram resultados positivos, cerca de 90%.

Além disso, foi possível entregar a suíte de testes automatizados em um arquivo com extensão específica para uso com a ferramenta Postman, permitindo assim que os testes sejam utilizados futuramente pela equipe responsável pelo Hawa sempre que necessário. Em conjunto com a suíte de testes automatizados, foi entregue um documento que deve ser utilizado como guia para uso dos testes automatizados com o Postman. É possível ter acesso a esse documento na íntegra no Apêndice D.

Por fim, sendo mais um resultado positivo do presente trabalho, foi entregue um documento baseado na documentação do Postman que pode ser utilizado como guia para a inclusão dos testes automatizados de integração implementados em futuras pipelines de validação. Também foi configurado e disponibilizado um repositório já preparado para execução dos testes automatizados implementados em uma pipeline de validação. É possível ter acesso a esse documento na íntegra no Apêndice E.

5.2 RESULTADOS NEGATIVOS

Poucos resultados negativos foram observados durante a execução do presente trabalho. Apenas 6 validações de um total de 63 demonstraram resultados preocupantes quanto ao estado da API REST, representando cerca de 10%. Ao analisar os resultados negativos obtidos em conjunto com a equipe responsável pelo Hawa, percebeu-se que os erros estavam relacionados à instabilidades no ambiente de teste e não ao software, fato que permitiu concluir que nenhum erro grave foi observado durante as validações.

Outro ponto negativo observado foi a percepção de que alguns dos endpoints da API REST do Hawa, devido ao contexto necessário para o uso dos mesmos, não são passíveis de automatização. Consequentemente esses endpoints precisam ser validados manualmente ou validados com auxílios de mocks.

Também foram percebidos alguns problemas com a documentação da API REST do software, pois como existem várias versões do produto, em alguns casos as documentações existentes não se mostraram integralmente corretas. Apesar de ser um ponto negativo, a oportunidade de melhorar as documentações do software surgiu como consequência.

6 CONSIDERAÇÕES FINAIS

Este trabalho descreveu os conceitos relacionados e a importância dos testes automatizados para o contexto atual da indústria de software. Para visualizar na prática os benefícios da utilização dos testes automatizados, uma suíte de testes automatizados de integração foi planejada e implementada para uso na validação da API REST pública do software Hawa, sendo este carente de testes automatizados de integração e utilizado para a emissão de certificados digitais.

Espera-se que essa suíte de testes automatizados possa contribuir para a melhoria constante da qualidade do software citado, consequentemente tornando a emissão de certificados digitais pelas entidades de hierarquia mais baixa na cadeia de entidades certificadoras da ICP - Brasil mais confiável e segura. Documentos complementares guiando o uso futuro da suíte de testes automatizados foram escritos para que esse objetivo seja alcançado com maior facilidade pela equipe responsável pelo Hawa.

Além dos pontos positivos citados, oportunidades de melhorias foram encontradas e reportadas, como melhoras na documentação do software e a criação de mocks que possam ser utilizados na validação automatizada do Hawa.

6.1 TRABALHOS FUTUROS

Oportunidades de melhorias foram percebidas analisando os pontos negativos do presente trabalho. Como primeira oportunidade de melhoria, é possível realizar correções e atualizações na documentação do software, garantindo assim um melhor entendimento por parte dos interessados no Hawa que tenham acesso aos documentos citados.

Também percebeu-se que alguns endpoints do Hawa, devido a características e contextos existentes, atualmente não são passíveis de validações automatizadas quando integrados a outros componentes. Desse fato surge a oportunidade de construir mocks que possam simular o funcionamento dos endpoints que não são passíveis de automação, diminuindo mais ainda a necessidade de testes manuais, fator interessante considerando o porte pequeno da equipe responsável pelo software e limitação de tempo existente.

Por fim, é possível configurar pipelines de validação automatizada para o Hawa, adicionando os próprios testes de integração automa-

tizados implementados durante o presente trabalho como componente da estrutura de validação. Um documento para auxílio nesse processo foi produzido e disponibilizado para a equipe do LabSEC responsável pelo Hawa.

7 DIREITOS AUTORAIS

O autor é o único responsável pelo conteúdo do material impresso incluído no seu trabalho.

REFERÊNCIAS

- FADEL, Aline Cristine; SILVEIRA, Henrique da Mota. *Metodologias ágeis no contexto de desenvolvimento de software: XP, Scrum e Lean*. Monografia do Curso de Mestrado FT-027-Gestão de Projetos e Qualidade da Faculdade de Tecnologia-UNICAMP, v. 98, p. 101, 2010.
- SEMEDO, Maria João Moreno *Ganhos de produtividade e de sucesso de Metodologias Ágeis VS Metodologias em Cascata no desenvolvimento de projectos de software*. 2012.
- BERNARDO, Paulo Cheque; KON, Fabio. *A importância dos Testes Automatizados*. Engenharia de Software Magazine, v. 1, n. 3, p. 54-57, 2008.
- ICP - BRASIL. ITI - Instituto Nacional de Tecnologia da Informação, Brasília, 27 de Junho de 2017. Disponível em: <<https://www.iti.gov.br/icp-brasil>>. Acesso em: 15 de Outubro de 2019.
- DELAMARO, Marcio; JINO, Mario; MALDONADO, Jose. *Introdução ao teste de software*. Elsevier Brasil, 2017.
- CRESPO, Adalberto Nobiato et al. *Uma metodologia para teste de Software no Contexto da Melhoria de Processo*. Simpósio Brasileiro de Qualidade de Software, p. 271-285, 2004.
- DOS SANTOS SOARES, Michel. *Comparação entre metodologias Ágeis e tradicionais para o desenvolvimento de software*. INFOCOMP Journal of Computer Science, v. 3, n. 2, p. 8-13, 2004.
- PRESSMAN, Roger; MAXIM, Bruce. *Engenharia de Software-8ª Edição*. McGraw Hill Brasil, 2016.
- DOS SANTOS SOARES, Michel. *Metodologias ágeis extreme programming e scrum para o desenvolvimento de software*. Revista Eletrônica de Sistemas de Informação, v. 3, n. 1, 2004.
- SILVA, Aldenara Moreira. *Um plano de garantia da qualidade para o software Promob*. 2016.
- MORAES, Lauriane Corrêa Pereira et al. *Um estudo empírico sobre o uso do BDD e seu apoio a engenharia de requisitos*. 2016.

CHIAVEGATTO, Rafael B. et al. *Desenvolvimento Orientado a Comportamento com Testes Automatizados utilizando JBehave e Selenium*. Anais do Encontro Regional de Computação e Sistemas de Informação Manaus. 2013.

ARAUJO, Mike Christian de Sousa. *Uma abordagem orientada a aspecto para escrita de história do usuário com Gherkin*. 2017. Dissertação de Mestrado. Universidade Federal de Pernambuco.

RODRIGUEZ, Alex. *Restful web services: The basics*. IBM Developer Works, v. 33, p. 18, 2008.

Site oficial da ferramenta Apache JMeter. Disponível em: <<https://jmeter.apache.org/index.html>>. Acesso em: 15 de Outubro de 2019.

Site oficial da ferramenta Katalon Studio. Disponível em: <<https://docs.katalon.com/katalon-studio/docs/index.html#products>>. Acesso em: 15 de Outubro de 2019.

Site oficial da ferramenta SoapUI. Disponível em: <<https://www.soapui.org/open-source.html>>. Acesso em: 15 de Outubro de 2019.

Site oficial da ferramenta REST-assured. Disponível em: <<http://rest-assured.io/>>. Acesso em: 15 de Outubro de 2019.

Site oficial da ferramenta Postman. Disponível em: <<https://www.getpostman.com/>>. Acesso em: 15 de Outubro de 2019.

Site oficial do Node.js. Disponível em: <<https://nodejs.org/en/>>. Acesso em 19 de Outubro de 2019.

SCHWABER, Ken; BEEDLE, Mike. *Agile software development with Scrum*. Upper Saddle River: Prentice Hall, 2002.

BECK, Kent. *Embracing change with extreme programming*. Computer, n. 10, p. 70-77, 1999.

BECK, Kent et al. *Manifesto for agile software development*. 2001.

ANDERSON, A., BEATTIE, R. e BECK, K. *Chrysler Goes to Extremes*. Disrupted Computers, 24-28, 1998.

BARTIÉ, Alexandre. *Garantia da qualidade de software*. Gulf Professional Publishing, 2002.

PRESSMAN, Roger S. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

SOMMERVILLE, Ian. *Software engineering*. Addison-Wesley/Pearson, 2011.

IEEE 829-2008 - IEEE Standard for Software and System Test Documentation. Disponível em: <<https://standards.ieee.org/standard/829-2008.html>>. Acesso em 30 de Novembro de 2019.

ITI - Instituto Nacional de Tecnologia da Informação. Disponível em: <<https://www.iti.gov.br/>>. Acesso em 30 de Novembro de 2019.

APÊNDICE A – Plano de teste

Testes de integração automatizados sobre API REST do software Hawa

Plano de Teste

Versão 1.0

Histórico da Revisão

Data	Versão	Descrição	Autor
21/Setembro/2019	1.0	Plano de Teste	Rodrigo Bittencourt de Lima

Índice

1. Objetivos
2. Escopo
3. Referências
4. Requisitos do Teste
5. Estratégia de Teste
6. Recursos
7. Produtos de Trabalho
8. Tarefas do Projeto

Plano de Teste

1. Objetivos

Este documento descreve o plano para a implementação de testes automatizados de integração sobre API REST do software Hawa. Este documento de Plano de Teste possui os seguintes objetivos:

- Identificar informações existentes do projeto e os componentes de software que devem ser testados.
- Listar os requisitos de teste recomendados.
- Descrever a estratégia de teste a ser empregada.
- Identificar os recursos necessários.
- Listar os elementos de produto de trabalho das tarefas de teste.

2. Escopo

Este Plano de Teste é aplicado aos testes automatizados de integração que serão implementados para a API REST do software Hawa. Assume-se que já existem testes de unidade aplicados sobre a API.

Este Plano de Teste é aplicado ao teste de todos os endpoints da API REST do software Hawa passíveis de validação automatizada, com base na documentação da API fornecida.

3. Referências

As referências aplicáveis são:

1. Exemplo da Web para Projeto de Registro em Curso, Versão 2001.03, IBM. Disponível em "http://mds.cultura.gov.br/extend_formal_resources/guidances/examples/resources/test_plan_v2.htm", acesso em 21 de Setembro de 2019.

4. Requisitos do Teste

A lista a seguir identifica os itens que foram identificados como alvos do teste. Essa lista representa o que será testado.

Testes de Integração automatizados de API REST

Validar todos os endpoints da API REST de forma automatizada.

Criar testes automatizados reutilizáveis.

Elaborar documento com instruções para uso futuro dos testes automatizados.

Elaborar documento com instruções para uso futuro dos testes automatizados em pipelines de validação.

5. Estratégia do Teste

A Estratégia de Teste apresenta a abordagem recomendada para o teste. A seção anterior dos Requisitos de Teste descreve o que será testado; esta descreve como será testado. As principais considerações para a estratégia de teste são as técnicas a serem utilizadas e o critério para saber quando o teste está concluído.

1. Tipos de Teste

1. Testes de Integração Automatizados

Os testes de integração automatizados devem ter foco na validação dos endpoints documentados da API REST do software Hawa. O objetivo desses testes é validar o correto funcionamento da API REST de acordo com as requisições enviadas para os endpoints da mesma. Esse tipo de teste baseia-se no envio de requisições com parâmetros diversos para os endpoints da API REST, validando o retorno dessas requisições. A seguir é identificado um esboço do teste:

Objetivo do Teste:	Validar o correto funcionamento da API REST ao receber diversas requisições em seus endpoints, com parâmetros variados.
Técnica:	Encaminhar requisições para os endpoints da API REST do software Hawa com diversos parâmetros, validando os retornos recebidos.
Crítérios de Conclusão:	Todos os testes planejados foram automatizados e executados.

2. Ferramentas

As seguintes ferramentas serão empregadas para o teste da API:

	Ferramenta	Versão
Testes de Integração Automatizados	Postman	7.7.0

6. Recursos

Esta seção apresenta os recursos recomendados para a implementação dos testes automatizados sobre a API REST do software Hawa.

1. Trabalhadores

Esta tabela mostra as premissas de equipe para as tarefas de teste.

Recursos Humanos

Trabalhador	Recursos Mínimos Recomendados	Responsabilidades Específicas/Comentários
Engenheiro de testes	Rodrigo Bittencourt de Lima	Planejamento, análise e implementação dos testes automatizados.

2. Software

A tabela a seguir descreve os recursos de software necessários para a implementação dos testes automatizados sobre API REST do software Hawa.

Recursos	
Recurso	Descrição
Ambiente de testes	Ambiente acessível e com o software Hawa em execução.
Documentação	Documentação da API REST do software Hawa.

7. Produtos de Trabalho

Os produtos de trabalho das tarefas de teste definidas neste Plano de Teste estão descritos na tabela a seguir.

Produtos de Trabalho	Proprietário
Plano de Teste	Rodrigo Bittencourt de Lima

Ambiente de Teste	Equipe de desenvolvimento do software Hawa
Documentação de API REST	Equipe de desenvolvimento do software Hawa
Casos de teste	Rodrigo Bittencourt de Lima
Scripts de Teste Automatizados	Rodrigo Bittencourt de Lima
Resultados de Testes	Rodrigo Bittencourt de Lima

1. Conjunto de Teste

O Conjunto de Teste definirá todos os casos de teste e os scripts de teste associados a cada caso de teste.

2. Resultados dos testes

Registros de execução dos testes automatizados, fornecidos à equipe responsável pelo software Hawa.

8. Tarefas do Projeto

A seguir são mostradas as tarefas relacionadas à implementação dos testes automatizados de integração sobre API REST do software Hawa:

Planejar Teste

Identificar Requisitos para o Teste

Desenvolver Estratégia de Teste

Identificar Recursos de Teste

Gerar Plano de Teste

Projetar Teste

Identificar e Descrever Casos de Teste

Identificar e Estruturar Scripts de Teste

Implementar Teste

Configurar Ambiente de Teste

Registrar ou Programar Scripts de
Teste

Executar Teste

Executar Script de Teste

Avaliar Execução do Teste

Verificar os Resultados

APÊNDICE B – Casos de teste

Testes de integração sobre API REST do software Hawa

Casos de Teste

Versão: 1.1

Data: 15/11/2019

Índice

1. PROPÓSITO.....	3
2. PÚBLICO ALVO.....	3
3. ESCOPO.....	3
4. DEFINIÇÕES E ABREVIACÕES	3
5. REFERÊNCIAS.....	3
6. CASOS DE TESTE.....	4

1. Propósito

Este documento contém os casos de teste que devem ser automatizados para validação da API REST do software Hawa.

2. Público Alvo

Este documento pode ser disponibilizado e analisado por todos os interessados nos testes automatizados, desde que autorizados previamente pela equipe responsável pelo software Hawa.

3. Escopo

Neste documento estão detalhados todos os casos de teste necessários para análise completa dos endpoints passíveis de validação da API REST do software Hawa.

4. Definições e abreviações

CT - caso de teste

5. Referências

Todos os casos de teste descritos neste documento foram desenhados com base na documentação da API REST do software Hawa.

6. Casos de Teste

6.1 [CT 001] - Criar certificate application com sucesso

Dado que usuário deseja criar um certificate application

Quando realiza uma requisição POST para o endpoint `/plugin/certificate-application/create`

E essa requisição possui o query param `"profile"` contendo uma String válida para o enum `"CertificateProfileEnum"`

E essa requisição possui em seu body um objeto `"CertificateApplicationDTO"` válido

Então deve receber um status http de retorno OK (200)

E deve receber um JSON de resposta contendo os campos `"protocol"`,

`"registrationAuthorityResponse"` e `"errorMessage"`

E o campo `"protocol"` deve conter o número de protocolo da aplicação criada

E o campo `"registrationAuthorityResponse"` deve conter o status http da RA

E o campo `"errorMessage"` deve conter uma mensagem confirmando a ausência de erros

6.2 [CT 002] - Criar certificate application com sucesso sem aprová-la

Dado que usuário deseja criar um certificate application

Quando realiza uma requisição POST para o endpoint `/plugin/certificate-application/create`

E essa requisição possui o query param `"profile"` contendo uma String válida para o enum `"CertificateProfileEnum"`

E essa requisição possui em seu body um objeto `"CertificateApplicationDTO"` válido

Então deve receber um status http de retorno OK (200)

E deve receber um JSON de resposta contendo os campos `"protocol"`,

`"registrationAuthorityResponse"` e `"errorMessage"`

E o campo `"protocol"` deve conter o número de protocolo da aplicação criada

E o campo `"registrationAuthorityResponse"` deve conter o status http da RA

E o campo `"errorMessage"` deve conter uma mensagem confirmando a ausência de erros

6.3 [CT 003] - "CertificateApplicationDTO" inválido

Dado que usuário deseja criar um certificate application

Quando realiza uma requisição POST para o endpoint `/plugin/certificate-application/create`

E essa requisição possui o query param `"profile"` contendo uma String válida para o enum `"CertificateProfileEnum"`

E essa requisição possui em seu body um objeto `"CertificateApplicationDTO"` inválido

Então deve receber um status http de retorno OK (200)

E deve receber um JSON de resposta contendo os campos `"protocol"`,

`"registrationAuthorityResponse"` e `"errorMessage"`

E o campo `"protocol"` deve conter o número de protocolo "0"

E o campo `"registrationAuthorityResponse"` deve conter o status http da RA

E o campo `"errorMessage"` deve conter uma mensagem informando o erro

6.4 [CT 004] - "Profile" desconhecido

Dado que usuário deseja criar um certificate application
Quando realiza uma requisição POST para o endpoint `"/plugin/certificate-application/create"`
E essa requisição possui o query param `"profile"` contendo uma String inválida para o enum `"CertificateProfileEnum"`
E essa requisição possui em seu body um objeto `"CertificateApplicationDTO"` válido
Então deve receber um status http de retorno BAD REQUEST (400)
E deve receber um JSON de resposta contendo o campo `"errorMessage"`
E o campo `"errorMessage"` deve conter uma mensagem informando o erro

6.5 [CT 005] - Aprovar certificate application

Dado que usuário deseja aprovar um certificate application
Quando realiza uma requisição POST para o endpoint `"/plugin/certificate-application/{protocol}/update"`
E essa requisição possui o query param `"rao-id"` contendo o ID do agente de registro
E essa requisição possui o query param `"ti-id"` contendo o ID da instalação técnica
E essa requisição possui o query param `"ca-id"` contendo o ID da entidade certificadora
E essa requisição possui o query param `"action"` contendo uma ação válida
E essa requisição possui em seu body um objeto `"CertificateApplicationDTO"` válido
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos `"status"`, `"authenticationCode"`, `"registrationAuthorityResponse"` e `"errorMessage"`
E o campo `"status"` deve conter o nova status da aplicação
E o campo `"authenticationCode"` deve conter o código de autenticação gerado para a aplicação
E o campo `"registrationAuthorityResponse"` deve conter o status http da RA
E o campo `"errorMessage"` deve conter uma mensagem confirmando a ausência de erros

6.6 [CT 006] - Busca por número do protocolo

Dado que usuário deseja encontrar um registro na base de aplicações informando o número do protocolo
Quando realiza uma requisição GET para o endpoint `"/plugin/certificate-application/search"`
E essa requisição possui o query param `"rao-id"` contendo o ID do agente de registro
E essa requisição possui o query param `"ti-id"` contendo o ID da instalação técnica
E essa requisição possui o query param `"type"` contendo o valor `"2"`
E essa requisição possui o query param `"value"` contendo um número de protocolo existente na base de aplicações
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos `"searchResult"`, `"registrationAuthorityResponse"` e `"errorMessage"`
E o campo `"searchResult"` deve conter um objeto `"ApplicationSearchResponse"` com os dados do registro encontrado
E o campo `"registrationAuthorityResponse"` deve conter o status http da RA
E o campo `"errorMessage"` deve conter uma mensagem confirmando a ausência de erros

6.7 [CT 007] - Busca por nome

Dado que usuário deseja encontrar um registro na base de aplicações informando o nome
Quando realiza uma requisição GET para o endpoint "/plugin/certificate-application/search"
E essa requisição possui o query param "rao-id" contendo o ID do agente de registro
E essa requisição possui o query param "ti-id" contendo o ID da instalação técnica
E essa requisição possui o query param "type" contendo o valor "0"
E essa requisição possui o query param "value" contendo um valor de nome existente na base de aplicações
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "searchResult", "registrationAuthorityResponse" e "errorMessage"
E o campo "searchResult" deve conter um objeto "ApplicationSearchResponse" com os dados do registro encontrado
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem confirmando a ausência de erros

6.8 [CT 008] - Busca por cpf

Dado que usuário deseja encontrar um registro na base de aplicações informando o cpf
Quando realiza uma requisição GET para o endpoint "/plugin/certificate-application/search"
E essa requisição possui o query param "rao-id" contendo o ID do agente de registro
E essa requisição possui o query param "ti-id" contendo o ID da instalação técnica
E essa requisição possui o query param "type" contendo o valor "1"
E essa requisição possui o query param "value" contendo um valor de cpf existente na base de aplicações
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "searchResult", "registrationAuthorityResponse" e "errorMessage"
E o campo "searchResult" deve conter um objeto "ApplicationSearchResponse" com os dados do registro encontrado
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem confirmando a ausência de erros

6.9 [CT 009] - Busca por registro inexistente

Dado que usuário deseja encontrar um registro na base de aplicações informando o nome
Quando realiza uma requisição GET para o endpoint "/plugin/certificate-application/search"
E essa requisição possui o query param "rao-id" contendo o ID do agente de registro
E essa requisição possui o query param "ti-id" contendo o ID da instalação técnica
E essa requisição possui o query param "type" contendo o valor "0"
E essa requisição possui o query param "value" contendo um valor de nome inexistente na base de aplicações
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "searchResult", "registrationAuthorityResponse" e "errorMessage"

E o campo "searchResult" deve conter um objeto "ApplicationSearchResponse" vazio
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem confirmando a ausência de erros

6.10 [CT 010] - "Type" não esperado

Dado que usuário deseja encontrar um registro na base de aplicações
Quando realiza uma requisição GET para o endpoint "/plugin/certificate-application/search"
E essa requisição possui o query param "rao-id" contendo o ID do agente de registro
E essa requisição possui o query param "ti-id" contendo o ID da instalação técnica
E essa requisição possui o query param "type" contendo o valor "10"
E essa requisição possui o query param "value" contendo um valor qualquer
Então deve receber um status http de retorno BAD REQUEST (400)
E deve receber um JSON de resposta contendo o campo "errorMessage"
E o campo "errorMessage" deve conter uma mensagem informando o erro

6.11 [CT 011] - "Value" incoerente para o "type" informado

Dado que usuário deseja encontrar um registro na base de aplicações informando o CPF
Quando realiza uma requisição GET para o endpoint "/plugin/certificate-application/search"
E essa requisição possui o query param "rao-id" contendo o ID do agente de registro
E essa requisição possui o query param "ti-id" contendo o ID da instalação técnica
E essa requisição possui o query param "type" contendo o valor "1"
E essa requisição possui o query param "value" contendo um valor inválido para CPF
Então deve receber um status http de retorno BAD REQUEST (400)
E deve receber um JSON de resposta contendo o campo "errorMessage"
E o campo "errorMessage" deve conter uma mensagem informando o erro

6.12 [CT 012] - Obter dados de aplicação informando número de protocolo

Dado que usuário deseja encontrar um registro na base de aplicações informando o protocolo
Quando realiza uma requisição GET para o endpoint
"/plugin/certificate-application/{numero_protocolo_aplicacao_X}"
E essa requisição possui o query param "code" contendo o código de autenticação da aplicação X
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "certificateApplication",
"registrationAuthorityResponse" e "errorMessage"
E o campo "certificateApplication" deve conter um objeto "CertificateApplicationResponse"
com os dados da aplicação X encontrada
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem confirmando a ausência de erros

6.13 [CT 013] - Busca por protocolo inexistente

Dado que usuário deseja encontrar um registro na base de aplicações informando o protocolo
Quando realiza uma requisição GET para o endpoint
"/plugin/certificate-application/{numero_protocolo_aplicacao_inexistente}"
E essa requisição possui o query param "code" contendo um código de autenticação válido
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "certificateApplication",
"registrationAuthorityResponse" e "errorMessage"
E o campo "certificateApplication" deve conter um objeto "CertificateApplicationResponse"
vazio
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem informando que não foi possível
encontrar resultados

6.14 [CT 014] - Busca por protocolo com código de autenticação inválido

Dado que usuário deseja encontrar um registro na base de aplicações informando o protocolo
Quando realiza uma requisição GET para o endpoint
"/plugin/certificate-application/{numero_protocolo_aplicacao_X}"
E essa requisição possui o query param "code" contendo um código de autenticação
inválido para a aplicação X
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "certificateApplication",
"registrationAuthorityResponse" e "errorMessage"
E o campo "certificateApplication" deve conter um objeto "CertificateApplicationResponse"
vazio
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem informando o erro

6.15 [CT 015] - Busca por certificado aprovado inexistente

Dado que usuário deseja emitir o certificado de um registro na base de aplicações
informando o protocolo
Quando realiza uma requisição POST para o endpoint
"/plugin/certificate-application/{numero_protocolo_aplicacao_inexistente}/certificate/issue"
E essa requisição possui o query param "code" contendo um código de autenticação válido
E essa requisição contém em seu body o parâmetro "PKCS10CertificationRequest" contendo
a requisição de certificado PKCS10 codificado em Base64
Então deve receber um status http de retorno OK (200)
E deve receber um JSON de resposta contendo os campos "certificatePEM",
"registrationAuthorityResponse" e "errorMessage"
E o campo "certificatePEM" deve conter uma string vazia
E o campo "registrationAuthorityResponse" deve conter o status http da RA
E o campo "errorMessage" deve conter uma mensagem informando que não foi possível
emitir o certificado desejado

6.16 [CT 016] - Busca por certificado aprovado com código de autenticação inválido

Dado que usuário deseja emitir o certificado de um registro na base de aplicações informando o protocolo

Quando realiza uma requisição POST para o endpoint

"/plugin/certificate-application/{numero_protocolo_aplicacao_X}/certificate/issue"

E essa requisição possui o query param "code" contendo um código de autenticação de inválido para a aplicação X

E essa requisição contém em seu body o parâmetro "PKCS10CertificationRequest" contendo a requisição de certificado PKCS10 codificado em Base64

Então deve receber um status http de retorno OK (200)

E deve receber um JSON de resposta contendo os campos "certificatePEM", "registrationAuthorityResponse" e "errorMessage"

E o campo "certificatePEM" deve conter uma string vazia

E o campo "registrationAuthorityResponse" deve conter o status http da RA

E o campo "errorMessage" deve conter uma mensagem informando que não foi possível emitir o certificado desejado

6.17 [CT 017] - Busca por certificado de aplicação ainda não aprovado

Dado que usuário deseja emitir o certificado de um registro na base de aplicações informando o protocolo

Quando realiza uma requisição POST para o endpoint

"/plugin/certificate-application/{numero_protocolo_aplicacao_ nao_aprovada}/certificate/issue"

E essa requisição possui o query param "code" contendo um código de autenticação inválido para a aplicação ainda não aprovada

E essa requisição contém em seu body o parâmetro "PKCS10CertificationRequest" contendo a requisição de certificado PKCS10 codificado em Base64

Então deve receber um status http de retorno OK (200)

E deve receber um JSON de resposta contendo os campos "certificatePEM", "registrationAuthorityResponse" e "errorMessage"

E o campo "certificatePEM" deve conter uma string vazia

E o campo "registrationAuthorityResponse" deve conter o status http da RA

E o campo "errorMessage" deve conter uma mensagem informando que não foi possível emitir o certificado desejado

APÊNDICE C – Código dos testes automatizados

Testes de integração sobre API REST do software Hawa

Código dos Testes Automatizados

Versão: 1.0
Data: 21/10/2019

Índice

1. PROPÓSITO.....	3
2. PÚBLICO ALVO.....	3
3. ESCOPO.....	3
4. ABREVIACÕES.....	3
5. REFERÊNCIAS.....	3
6. CÓDIGO DOS TESTES AUTOMATIZADOS.....	4
7. CÓDIGO DO ARQUIVO DE CONFIGURAÇÃO NO GITLAB.....	12

1. Propósito

Este documento contém o código dos testes automatizados implementados durante a execução do trabalho proposto. Nele são detalhadas as requisições, os parâmetros das requisições, as validações de resposta e as variáveis globais utilizadas.

2. Público Alvo

Este documento pode ser disponibilizado e analisado por todos os interessados nos testes automatizados, desde que autorizados previamente pela equipe responsável pelo software Hawa.

3. Escopo

Neste documento são exibidos os detalhes técnicos dos testes automatizados implementados. Os testes são apresentados na ordem em que são executados durante a suíte de testes.

4. Abreviações

EP = endpoint

CT = Caso de teste

5. Referências

Todos os testes foram implementados com base na documentação oficial do software Hawa. A documentação não está disponível para acesso público.

6. Código dos testes automatizados

EP - /plugin/certificate-application/create

CT 1 - Criar certificate application com sucesso

URL: [URL_INICIAL_HAWA/certificate-application/create](#)

Query params: *profile = NATURAL_PERSON*

Body: JSON com dados para o objeto CertificateApplication

Código dos testes que validam a resposta:

```
1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns the protocol number of the created application', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.protocol).to.be.at.least(1);
8 });
9
10 pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("201 Created");
13 });
14
15 pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });
19
20 let jsonData = pm.response.json();
21 pm.environment.set("protocol", jsonData.protocol);
```

CT 2 - Criar certificate application com sucesso sem aprová-la

URL: [URL_INICIAL_HAWA/certificate-application/create](#)

Query params: *profile = NATURAL_PERSON*

Body: JSON com dados para o objeto CertificateApplication

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns the protocol number of the created application', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.protocol).to.be.at.least(1);
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("201 Created");
13 });
14
15 - pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });
19
20 let jsonData = pm.response.json();
21 pm.environment.set("unapprovedProtocol", jsonData.protocol);

```

CT 3 - "CertificateApplicationDTO" inválido

URL: URL_INICIAL_HAWA/certificate-application/create

Query params: *profile* = NATURAL_PERSON

Body: JSON com dados **inválidos** para o objeto CertificateApplication

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns the protocol zero', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.protocol).to.eql(0);
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("304 Not Modified");
13 });
14
15 - pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("certificate application could not be created");
18 });

```

CT 4 - "profile" desconhecido

URL: URL_INICIAL_HAWA/certificate-application/create

Query params: *profile* = INVALID_PROFILE

Body: JSON com dados para o objeto CertificateApplication

Código dos testes que validam a resposta:

```

1 pm.test('returns a 400 response', () => {
2   pm.response.to.have.status(400);
3 });
4
5 pm.test('returns the correct error message', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.errorMessage).to.eql("Unknown profile");
8 });

```

EP - /plugin/certificate-application/update

CT 5 - Aprovar certificate application

URL: URL_INICIAL_HAWA/certificate-application/{protocol}/update

Query params: `rao-id=1 & ti-id=1 & ca-id=1 & action=APPROVED`

Body: JSON com dados para o objeto CertificateApplication

Código dos testes que validam a resposta:

```

1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns the new application status', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.status).to.eql("approved");
8 });
9
10 pm.test('returns the authentication code', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.authenticationCode).not.eql("undefined");
13 });
14
15 pm.test('returns the correct registration authority response', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("202 Accepted");
18 });
19
20 pm.test('returns no error message', () => {
21   let jsonData = pm.response.json();
22   pm.expect(jsonData.errorMessage).to.eql("no error");
23 });
24
25
26 let jsonData = pm.response.json();
27 pm.environment.set("authenticationCode", jsonData.authenticationCode);

```

EP - /plugin/certificate-application/search

CT 6 - Busca por número do protocolo

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=2 & value={{protocol}}*

Código dos testes que validam a resposta:

```
1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns the correct search result', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.searchResult.applicationList[0].certificateApplication.id).to.eql(pm.environment
8     .get("protocol"));
9 });
10 pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("200 OK");
13 });
14
15 pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });
```

CT 7 - Busca por nome

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=0 & value=NOME1*

Código dos testes que validam a resposta:

```
1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns the correct search result', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.searchResult.applicationList[0].certificateApplication.name).to.eql("NOME1");
8 });
9
10 pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("200 OK");
13 });
14
15 pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });
```

CT 8 - Busca por CPF

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=1 & value=26974640005*

Código dos testes que validam a resposta:

```

1 ▾ pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 ▾ pm.test('returns the correct search result', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.searchResult.applicationList[0].certificateApplication.cpf).to.eql("26974640005"
8   );
9 });
10 ▾ pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("200 OK");
13 });
14
15 ▾ pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });

```

CT 9 - Busca por registro inexistente

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=0 & value=namenexistente*

Código dos testes que validam a resposta:

```

1 ▾ pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 ▾ pm.test('does not returns results', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.searchResult.applicationList.length).to.eql(0);
8 });
9
10 ▾ pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("200 OK");
13 });
14
15 ▾ pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });

```

CT 10 - "Type" não esperado

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=10 & value=any*

Código dos testes que validam a resposta:

```

1 pm.test('returns a 400 response', () => {
2   pm.response.to.have.status(400);
3 });
4
5 pm.test('returns the correct error message', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.errorMessage).to.eql("Invalid search type");
8 });

```

CT 11 - "Value" incoerente para o "type" informado

URL: URL_INICIAL_HAWA/certificate-application/search

Query params: *rao-id=1 & ti-id=1 & type=1 & value=cpfInvalido*

Código dos testes que validam a resposta:

```

1 pm.test('returns a 400 response', () => {
2   pm.response.to.have.status(400);
3 });
4
5 pm.test('returns the correct error message', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.errorMessage).to.eql("malformed CPF expression");
8 });

```

EP - /plugin/certificate-application/{protocol}

CT 12 - Obter dados de aplicação informando número de protocolo

URL: URL_INICIAL_HAWA/certificate-application/{protocol}

Query params: *code = {{authenticationCode}}*

Código dos testes que validam a resposta:


```

1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns the correct certificate application', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificateApplication.name).to.eql("NOME1");
8 });
9
10 pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("200 OK");
13 });
14
15 pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("no error");
18 });
19
20

```

CT 13 - Busca por registro inexistente

URL: URL_INICIAL_HAWA/certificate-application/999

Query params: `code = {{authenticationCode}}`

Código dos testes que validam a resposta:

```

1 pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test('returns empty certificate application', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificateApplication).to.eql({});
8 });
9
10 pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("204 No Content");
13 });
14
15 pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("Could not retrieve the desired certificate application,
18     parameters may be wrong");
19 });
20

```

CT 14 - Busca por registro com código de autenticação inválido

URL: URL_INICIAL_HAWA/certificate-application/{{protocol}}

Query params: `code = invalidAuthCode`

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns empty certificate application', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificateApplication).to.eql({});
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("204 No Content");
13 });
14
15 - pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("Could not retrieve the desired certificate application,
    parameters may be wrong");
18 });

```

EP - /plugin/certificate-application/{protocol}/certificate/issue

CT 15 - Busca por registro inexistente

URL: URL_INICIAL_HAWA/certificate-application/999/certificate/issue

Query params: *code* = {{authenticationCode}}

Body: JSON com string contendo a requisição de certificado PKCS10 codificado em Base64

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns empty certificate PEM', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificatePEM).to.eql("");
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("304 Not Modified");
13 });
14
15 - pm.test('returns no error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("Could not issue the desired certificate");
18 });

```

CT 16 - Busca por registro com código de autenticação inválido

URL: URL_INICIAL_HAWA/certificate-application/{protocol}/certificate/issue

Query params: *code* = *invalidAuthCode*

Body: JSON com string contendo a requisição de certificado PKCS10 codificado em Base64

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns empty certificate PEM', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificatePEM).to.eql("");
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("304 Not Modified");
13 });
14
15 - pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("Could not issue the desired certificate");
18 });

```

CT 17 - Busca por registro de aplicação ainda não aprovada

URL: [URL_INICIAL_HAWA/certificate-application/{\(unapprovedProtocol\)}/certificate/issue](URL_INICIAL_HAWA/certificate-application/{(unapprovedProtocol)}/certificate/issue)

Query params: *code = unapproved*

Body: JSON com string contendo a requisição de certificado PKCS10 codificado em Base64

Código dos testes que validam a resposta:

```

1 - pm.test('returns a 200 response', () => {
2   pm.response.to.have.status(200);
3 });
4
5 - pm.test('returns empty certificate PEM', () => {
6   let jsonData = pm.response.json();
7   pm.expect(jsonData.certificatePEM).to.eql("");
8 });
9
10 - pm.test('returns the correct registration authority response', () => {
11   let jsonData = pm.response.json();
12   pm.expect(jsonData.registrationAuthorityResponse.status).to.eql("304 Not Modified");
13 });
14
15 - pm.test('returns the correct error message', () => {
16   let jsonData = pm.response.json();
17   pm.expect(jsonData.errorMessage).to.eql("Could not issue the desired certificate");
18 });

```

7. Código do arquivo de configuração no gitlab

```
1 variables:
2   POSTMAN_COLLECTION: https://api.getpostman.com/collections/25
3   POSTMAN_ENVIRONMENT: https://api.getpostman.com/environments/
4
5 stages:
6 - postman
7
8 postman_job:
9   stage: postman
10  image: wojciechzurek/newman-ci
11  script:
12    - newman run ${POSTMAN_COLLECTION} -e ${POSTMAN_ENVIRONMENT}
```


APÊNDICE D - Guia para uso dos testes automatizados

Testes de integração sobre API REST do software Hawa

Guia para Uso dos Testes Automatizados

Versão: 1.0
Data: 19/10/2019

Índice

1. PROPÓSITO.....	3
2. PÚBLICO ALVO.....	3
3. ESCOPO.....	3
4. REFERÊNCIAS.....	3
5. INSTALAÇÃO DO POSTMAN.....	4
6. IMPORTAÇÃO DOS TESTES AUTOMATIZADOS.....	6
7. EXECUÇÃO DOS TESTES AUTOMATIZADOS.....	8
8. COLETA DOS RESULTADOS.....	11

1. Propósito

Este documento contém as instruções necessárias para a utilização dos testes automatizados de integração implementados para o software Hawa. As informações descritas aqui são úteis para uso dos testes em conjunto com a ferramenta Postman.

2. Público Alvo

Este documento pode ser disponibilizado e analisado por todos os interessados nos testes automatizados, desde que autorizados previamente pela equipe responsável pelo software Hawa.

3. Escopo

Neste documento estão detalhadas as ações necessárias para a execução dos testes de integração automatizados, implementados com o intuito de validar por completo todos os endpoints passíveis de validação automatizada da API REST do software Hawa. As informações descritas aqui são úteis para uso dos testes em conjunto com a ferramenta Postman.

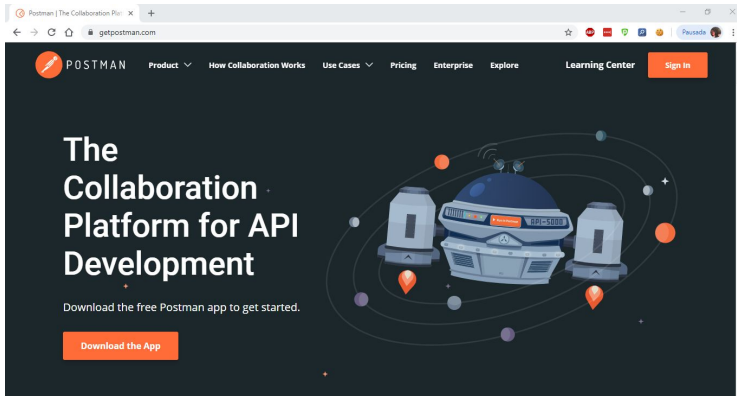
4. Referências

Site oficial da ferramenta Postman. Disponível em: <<https://www.getpostman.com/>>. Acesso em: 19 de Outubro de 2019.

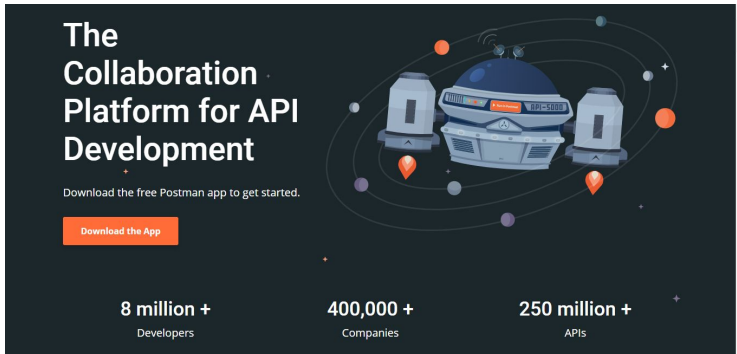
5. Instalação do Postman

Os passos descritos a seguir detalham as ações necessárias para a instalação correta da ferramenta Postman em sistema operacional Windows. Para instalação da ferramenta em outros sistemas operacionais, consultar a documentação oficial da ferramenta.

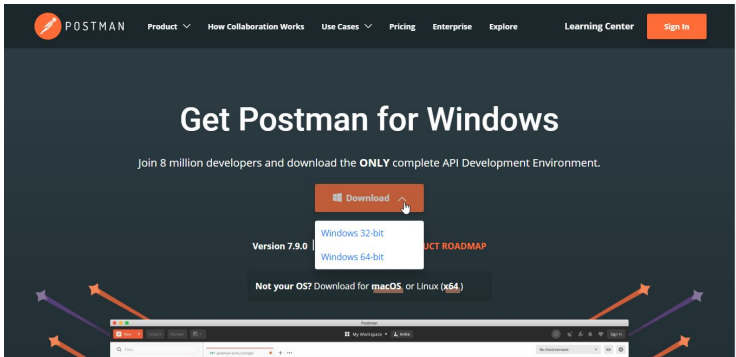
1. Acesse o site oficial da ferramenta (<https://www.getpostman.com/>)



2. Acione a opção "Download the App"



3. Selecione a versão correta para o sistema operacional em uso

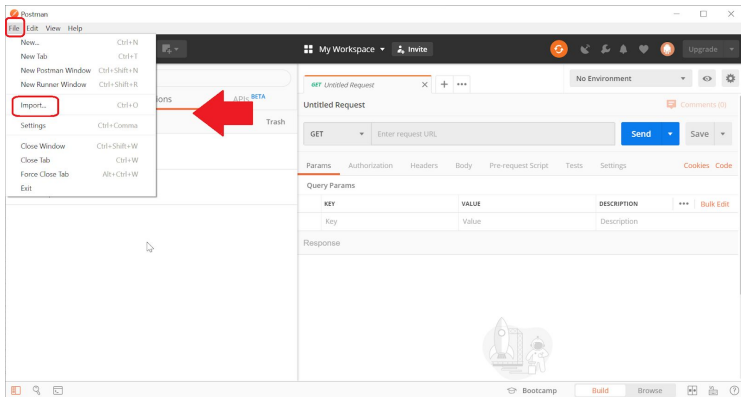


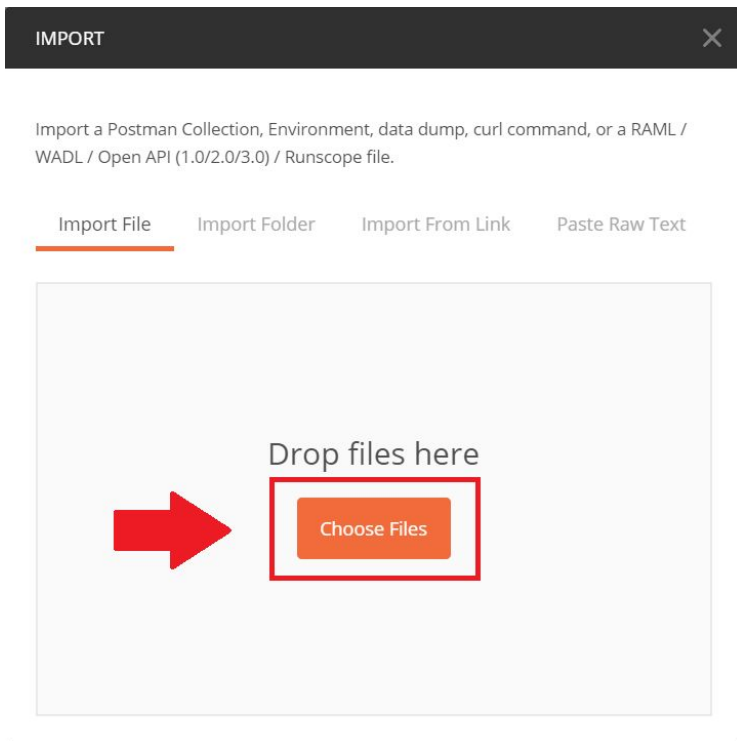
4. Execute o arquivo baixado e siga os passos de instalação

6. Importação dos testes automatizados

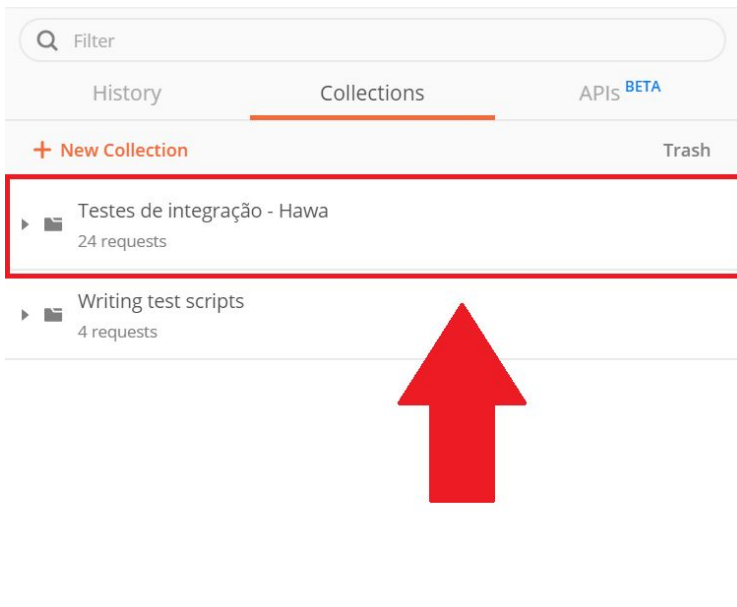
Os passos descritos a seguir detalham as ações necessárias para importar para o Postman o arquivo contendo os testes automatizados implementados. É necessário já possuir a ferramenta instalada.

1. Selecionar para importação o arquivo contendo os testes automatizados





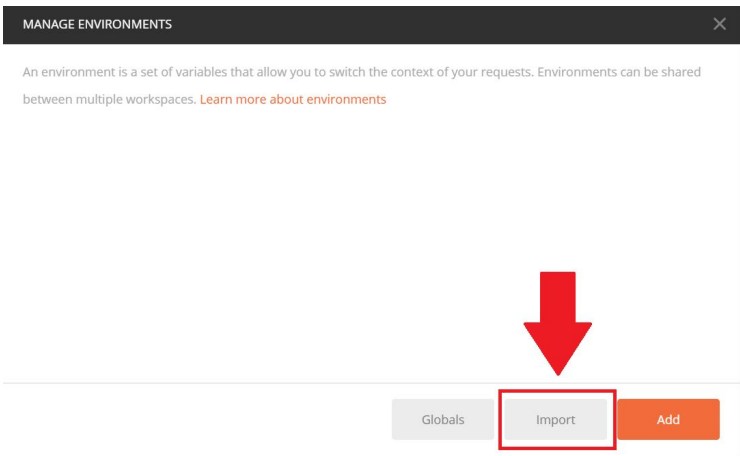
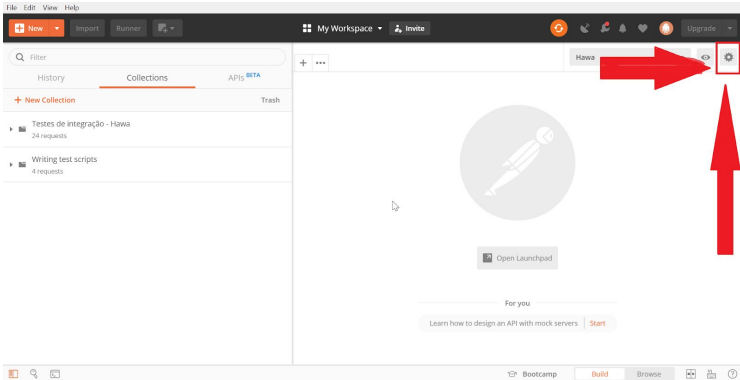
2. Após importar o arquivo, a suíte de testes estará disponível na aba "Collections"



7. Execução dos testes automatizados

Os passos descritos a seguir detalham as ações necessárias para a execução dos testes automatizados importados anteriormente para o Postman . É necessário já possuir a ferramenta instalada e o arquivo contendo os testes já importado.

1. Importar arquivo de environment para utilizar variáveis globais necessárias



MANAGE ENVIRONMENTS ✕

An environment is a set of variables that allow you to switch the context of your requests. Environments can be shared between multiple workspaces. [Learn more about environments](#)

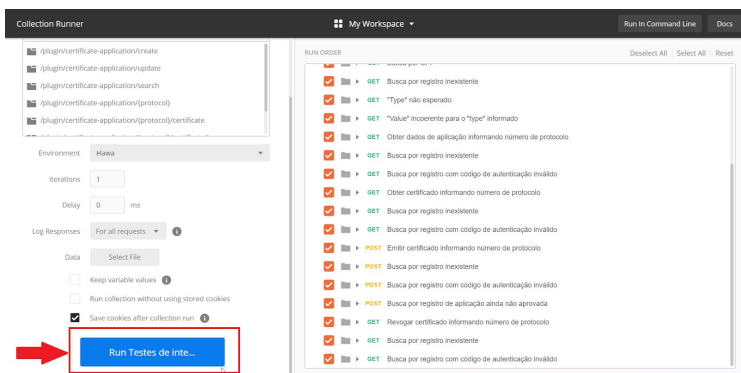
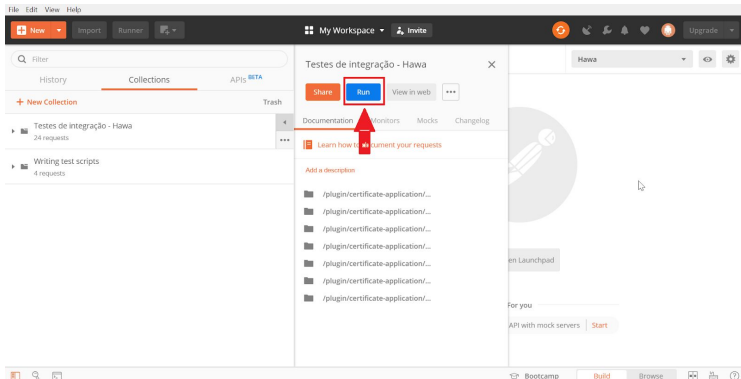
Hawa ➔ Share 📄 📥 ⋮



Globals Import Add

2. Executar a suíte de testes automatizados

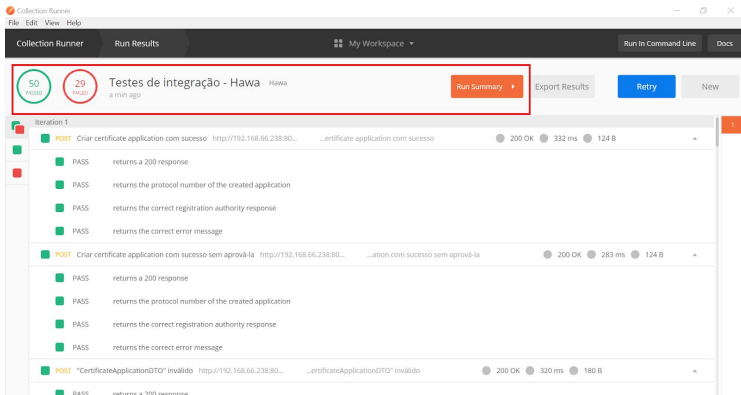
The screenshot shows the Postman application interface. On the left sidebar, under 'Collections', there is a collection named 'Testes de integração - Hawa' with 24 requests. Below it is 'Writing test scripts' with 4 requests. A red box highlights the trash icon next to the 'Testes de integração - Hawa' collection, and a red arrow points to it from the left. The main workspace area shows the 'Hawa' environment selected. The bottom status bar includes 'Bootcamp', 'Build', and 'Browse' buttons.



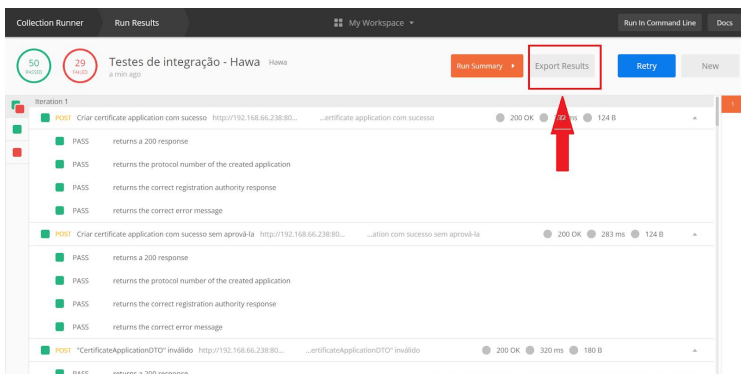
8. Coleta dos resultados

Os passos descritos a seguir detalham as ações necessárias para coletar os resultados de uma execução dos testes automatizados utilizando a ferramenta Postman. É necessário já ter realizado uma execução da suite de testes.

1. Tela com a exibição dos resultados



2. Exportação dos resultados



**APÊNDICE E - Guia para uso dos testes automatizados em
pipelines de validação**

Testes de integração sobre API REST do software Hawa

Guia para Adicionar os Testes Automatizados em Pipelines de Validação

Versão: 1.0
Data: 22/10/2019

Índice

1. PROPÓSITO.....	3
2. PÚBLICO ALVO.....	3
3. ESCOPO.....	3
4. REFERÊNCIAS.....	3
5. CONFIGURAÇÕES NECESSÁRIAS.....	4

1. Propósito

Este documento consiste em um guia rápido que visa orientar a execução dos passos necessários para adicionar os testes automatizados de integração implementados para o software Hawa em pipelines de validação.

2. Público Alvo

Este documento pode ser disponibilizado e analisado por todos os interessados nos testes automatizados, desde que autorizados previamente pela equipe responsável pelo software Hawa.

3. Escopo

Neste documento são abordados os passos necessários para adicionar os testes automatizados de integração implementados para o Hawa em uma pipeline de validação configurada para execução na ferramenta Gitlab CI. Optou-se por focar nessa ferramenta pois é a mais adequada para o contexto da equipe responsável pelo Hawa.

4. Referências

Run Postman tests with Newman in Gitlab CI. Disponível em:
<<https://medium.com/@autumn.bom/run-postman-tests-with-newman-in-gitlab-ci-c34bc90e17ec>>. Acesso em: 24 de Outubro de 2019.

5. Configurações necessárias

São necessários 4 passos para configurar a execução dos testes em uma pipeline do Gitlab CI, sendo eles: obter uma chave de API do Postman; obter a URL da coleção de testes (suíte de testes); obter a URL do ambiente de execução; configurar a pipeline no Gitlab.

1º Obter uma chave da API do Postman:

Para obter uma chave da API do Postman, basta abrir o painel web da ferramenta e copiar a chave obtida através do caminho "Web > Choose workspace > Tab Integrations", posteriormente acessando "Browse Integrations > Choose Postman API", e por fim acionando a função "Get API Key".

2º Obter a URL da coleção de testes (suíte de testes):

Para obter a URL da coleção de testes, primeiramente se faz necessário obter o identificador da coleção. Para isso, basta encaminhar uma request contendo a chave da API obtida anteriormente, para o seguinte endpoint:

"https://api.getpostman.com/collections?apikey=\${apiKey}", onde "apiKey" seria a chave da API.

Após o primeiro passo concluído, basta encaminhar outra request para "https://api.getpostman.com/collections/\${c_uid}?apikey=\${apiKey}", onde "c_uid" é o identificador da coleção obtido anteriormente. Executando a segunda request, será obtida a URL da coleção desejada.

3º Obter a URL do ambiente de execução:

Para obter a URL do ambiente de execução (obtendo assim as variáveis globais utilizadas durante a execução dos testes), primeiramente se faz necessário obter o identificador do ambiente encaminhando uma request para

"https://api.getpostman.com/environments?apikey=\${apiKey}", onde "apiKey" é a chave da API do Postman obtida anteriormente.

Após isso, basta encaminhar outra request para

"https://api.getpostman.com/environments/\${e_uid}?apikey=\${apiKey}", onde "e_uid" é o identificador do ambiente de execução. Executando a segunda request, será obtida a URL do ambiente de execução desejado.

4º Configurar a pipeline no Gitlab:

Finalizando, para configurar a pipeline em um repositório do Gitlab, basta adicionar o arquivo ".gitlab-ci.yml" com a configuração abaixo, fazendo com que os testes sejam executados a cada nova alteração de código adicionada ao repositório:

```
variables:
  POSTMAN_COLLECTION: https://api.getpostman.com/collections/${c_uid}?apikey=${apiKey}
  POSTMAN_ENVIRONMENT: https://api.getpostman.com/environments/${e_uid}?apikey=${apiKey}

stages:
- some_stages
- postman

postman_job:
  stage: postman
  image: wojciechzurek/newman-ci
  script:
  - newman run ${POSTMAN_COLLECTION} -e ${POSTMAN_ENVIRONMENT}
```


APÊNDICE F – Artigo

Testes de integração sobre API REST do software Hawa

Rodrigo Bittencourt de Lima

¹Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

Abstract. *With the increasing adoption of agile methodologies in software development, testing automation is becoming increasingly important. The purpose of this paper is to ensure the correct functioning of Hawa's REST API, which is a software whose purpose is to issue digital certificates to end users. Hawa is used by the lower hierarchy entities in the chain of ICP - Brazil certifying entities. To achieve the proposed goal, automated integration tests on Hawa's REST API were implemented and performed using current tools and technologies.*

Resumo. *Com a crescente adoção de metodologias ágeis no desenvolvimento de software, cada vez mais a automatização de testes ganha importância. O objetivo do presente trabalho é garantir o correto funcionamento da API REST do Hawa, sendo este um software cuja finalidade é a emissão de certificados digitais a usuários finais. O Hawa é utilizado pelas entidades de hierarquia mais baixa na cadeia de entidades certificadoras do ICP - Brasil. Para alcançar o objetivo proposto, foram implementados e executados testes automatizados de integração sobre a API REST do Hawa, utilizando ferramentas e tecnologias atuais.*

1. Introdução

A indústria de software busca continuamente por modelos de desenvolvimento de software cada vez mais eficientes. As novas metodologias disputam espaço com os modelos tradicionais de desenvolvimento. As metodologias ágeis se mostram eficientes, possuindo maior destaque na atualidade (FADEL; SILVEIRA, 2010).

As metodologias ágeis mostram-se efetivas na redução de custos. Para que as metodologias ágeis sejam capazes de aumentar a eficiência do processo de desenvolvimento, profissionais e atividades rotineiramente executadas durante esse processo precisam de adaptações. O processo de garantia de qualidade de software vem sofrendo mudanças constantemente (SEMEDO, 2012).

Com a adoção pelas metodologias ágeis de ciclos curtos de entrega e feedback com cliente, testes manuais de software se mostram caros e exaustivos. Tornou-se necessário pensar em soluções para adaptar o processo de garantia de qualidade às metodologias de desenvolvimento emergentes. A utilização de testes automatizados de regressão mostrou-se uma solução viável e promissora (BERNARDO; KON, 2008).

O presente trabalho surgiu da necessidade de garantia de qualidade automatizada sobre a API REST do software Hawa. Este produto é de propriedade do LabSEC, e sua

finalidade é a emissão de certificados digitais a usuários finais. O Hawa é utilizado pelas entidades pertencentes à cadeia certificadora da ICP – Brasil, que consiste em uma estrutura hierárquica que permite a emissão de certificados digitais para identificação virtual do cidadão por meio de uma cadeia de confiança.

A equipe de desenvolvimento do Hawa já faz o uso de testes automatizados para validação do produto, porém existe a necessidade de garantir que a API REST pública do software funcione corretamente e de forma integrada. Tomaram-se necessárias a modelagem e a implementação de uma suíte de testes automatizados de integração externamente à equipe, gerando a oportunidade de executar este trabalho.

2. Conceitos

2.1. Metodologias ágeis de desenvolvimento de software

A engenharia de software busca constantemente por oportunidades de maior lucratividade. As empresas estão sempre à procura de metodologias de desenvolvimento que definam processos capazes de resultarem em maior eficiência no uso de tempo e de recursos.

As metodologias ágeis de desenvolvimento surgiram como uma possibilidade de criar produtos de maior qualidade e capazes de atender as expectativas dos clientes. O processo de desenvolvimento definido por essas metodologias permite que o software seja entregue no prazo correto e com alta eficiência no uso de recursos. O fluxo de atividades extremamente organizado definido pelas metodologias ágeis mostra-se efetivo (FADEL e SILVEIRA, 2010).

O termo "Metodologias ágeis" tornou-se popular a partir do ano 2001, quando dezessete especialistas focados nos processos de desenvolvimento de software estabeleceram características e princípios que eram compartilhados pelas metodologias de desenvolvimento emergentes, definindo e criando o chamado "Manifesto Ágil" e a Aliança Ágil (SOARES, 2004a). Existem quatro princípios chave que são:

- Indivíduos e interações em vez de processos e ferramentas.
- Software executável em vez de documentação.
- Colaboração do cliente em vez de negociação de contratos.
- Respostas rápidas a mudanças em vez de seguir planos.

As metodologias ágeis se adaptam melhor às empresas de pequeno e médio porte, trazendo maior eficiência, lucratividade e capacidade de adaptação à mudanças. Scrum e XP (Extreme Programming) podem ser consideradas duas das metodologias ágeis mais conhecidas (SOARES, 2004a).

Várias metodologias ágeis citam a importância dos testes no desenvolvimento de um software. Muitas técnicas de testes foram criadas e evoluídas. Existem muitos tipos de testes, automatizados ou não (FADEL e SILVEIRA, 2010).

2.2. Testes de software

DELAMARO et al. (2016) afirmam que o processo envolvido na execução de testes de software é complexo, pois existem muitas variáveis que podem implicar na ocorrência de erros e falhas em um software. A atividade de testes precisou ser dividida em diferentes etapas e técnicas, capazes de validar todas as camadas de um produto. Os autores citam como principais fases o teste de unidade, o teste de integração e o teste de sistemas.

Os testes de regressão ou regressivos possuem como objetivo principal garantir que o software continua funcionando após novas funcionalidades ou alterações serem inseridas. É comum que um software possua problemas em diferentes unidades após uma delas sofrer alterações, ou após uma nova unidade ser integrada ao sistema. Por isso os testes de regressão mostram-se indispensáveis.

Quando pensamos nas metodologias ágeis, podemos ligar os testes de regressão com as entregas constantes e o fluxo iterativo de desenvolvimento. Em processos de desenvolvimento que utilizam metodologias ágeis, os testes regressivos são constantes e podem se tornar caros, isso porque geram retrabalho. Como consequência desse problema, os testes automatizados ganharam extrema importância no uso das metodologias ágeis.

2.3. Testes automatizados de software

Testes automatizados podem ser conceituados como rotinas que, com pouca ação humana envolvida, podem simular várias funcionalidades presentes em um software, validando todas as respostas e possíveis erros gerados como resultados. Como a interação humana é muito pequena, geralmente é possível executar uma suíte de testes automatizados muitas vezes de forma fácil e barata, além de pouco sujeita a falha humana já que a maior parte da atividade é executada por computadores.

Além de menos suscetíveis a falhas humanas, testes automatizados podem simular situações extremamente complexas por serem executados por computadores. É possível validar uma quantidade muito superior de combinações e contextos do que seria possível manualmente BERNARDO e KON (2008).

2.4. API REST

A arquitetura REST, modelo utilizado na estruturação de serviços Web, consiste em um conjunto de princípios e conceitos que guiam os desenvolvedores durante a construção de um software para a Web. Essa arquitetura é orientada aos recursos presentes no sistema em construção, definindo seus possíveis estados, como devem ser endereçados, e como devem ser compartilhados por meio do protocolo HTTP (Hypertext Transfer Protocol).

O modelo REST é considerado o predominante entre os modelos existentes para construção de serviços Web. Isso é observado por meio da quantidade de softwares desenvolvidos com uso dessa arquitetura. Por ser um modelo mais simples e eficiente, o REST substituiu os modelos SOAP e WSDL, que eram os mais utilizados antes do seu

surgimento RODRIGUEZ (2008). Um serviço Web construído com base na arquitetura REST deve seguir quatro princípios:

- Uso de métodos HTTP.
- Uso de requisições independentes (stateless).
- Uso de URI's baseadas na estrutura do projeto.
- Uso de XML, JavaScript Object Notation (JSON), ou ambos para transferência de dados.

3. Planejamento

Este capítulo apresenta todas as análises e definições realizadas previamente à implementação dos testes automatizados. São detalhados os processos de elaboração do plano de teste e dos casos de teste, e a definição de tecnologias e ferramentas a serem utilizadas.

3.1. Plano de teste

Plano de teste consiste em um documento que apresenta o planejamento para execução do teste, incluindo a abrangência, abordagem, recursos e cronograma das atividades de teste. Identifica os itens e as funcionalidades a serem testados, as tarefas a serem realizadas e os riscos associados com a atividade de teste (Norma IEEE 829).

Foi elaborado um plano de teste para guiar a implementação dos testes automatizados de integração. O mesmo foi adaptado ao contexto do presente trabalho.

3.2. Especificação dos Casos de teste

Especificação dos Casos de teste consiste em um documento que define os casos de teste, incluindo dados de entrada, resultados esperados, ações e condições gerais para a execução do teste (Norma IEEE 829).

Foi elaborado um documento de Especificação dos casos de teste para guiar a implementação dos testes automatizados. Tal documento foi desenvolvido com base na documentação da API Rest do software Hawa. Os casos de teste foram adaptados ao contexto e as necessidades do presente trabalho.

3.3. Definição de tecnologias e ferramentas

Para a implementação dos testes automatizados de integração sobre API REST do software Hawa, objetivo principal do presente trabalho, se faz necessário o uso de alguma ferramenta capaz de executar tal tipo de teste. Foram analisadas, levando em consideração a utilização atual no mercado e experiências prévias dos envolvidos no trabalho, cinco ferramentas, sendo elas: Apache JMeter, Katalon Studio, SoapUI, Rest-Assured e Postman.

Após testes realizados com cada uma das 5 ferramentas citadas anteriormente, verificou-se que todas são poderosas o suficiente para a execução com eficácia do trabalho proposto. Decidiu-se por utilizar o Postman como ferramenta de

implementação e execução dos testes automatizados sobre API REST. O Postman mostrou-se uma ferramenta confiável e eficiente.

4. Implementação

Este capítulo apresenta o processo de implementação dos testes automatizados. São apresentados o ambiente de testes utilizado e o processo de implementação dos testes automatizados utilizando a ferramenta Postman.

4.1. Ambiente de testes

Para que seja possível a implementação e a execução dos testes automatizados sobre API Rest do software Hawa, faz-se necessário a disponibilização de um ambiente de testes. Este ambiente deve possuir uma instância do software Hawa em execução e acessível remotamente.

Devido a essa necessidade, a equipe responsável pelo software Hawa configurou e disponibilizou uma versão do mesmo em execução em servidor com acesso remoto liberado. Dessa forma, foi possível implementar e executar diversas vezes os testes automatizados, realizando ajustes até que os mesmos se encontrassem na forma ideal.

4.2. Testes automatizados com Postman

Todo o processo de implementação dos testes automatizados foi realizado com base na documentação oficial da ferramenta Postman. A funcionalidade do Postman que possibilitou a utilização do mesmo no presente trabalho, foi a capacidade de execução de scripts por meio de uma poderosa ferramenta baseada em Node.js.

Ao configurar uma requisição no Postman, é possível adicionar scripts JavaScript para execução antes do envio da requisição, como também após o recebimento da resposta da mesma. Os testes automatizados são executados após o recebimento da resposta de uma requisição enviada com o Postman.

Para cada caso de teste desenhado na etapa de planejamento do presente trabalho, foi configurada uma requisição no Postman. Para cada requisição configurada, um script de teste foi codificado para validar a resposta recebida ao enviar a requisição.

5. Resultados

Este capítulo apresenta os resultados obtidos com a implementação e execução dos testes automatizados de integração sobre o software Hawa.

4.2. Resultados positivos

Após a automatização dos 17 casos de teste, totalizando cerca de 60 itens validados, sendo essa implementação realizada de forma colaborativa em conjunto com a equipe responsável pelo Hawa para que as validações estivessem de acordo com o esperado, a maioria dos resultados foram positivos.

Executando a suíte de casos de teste ao final da implementação, foi possível concluir que os endpoints da API REST pública do Hawa se comportam conforme o esperado para a maioria das validações, tanto em testes utilizando dados válidos como em contextos envolvendo dados inválidos. Isso vale para todos os endpoints passivos de

automatização. De 63 validações realizadas, 57 tiveram resultados positivos, cerca de 90%.

Além disso, foi possível entregar a suíte de testes automatizados em um arquivo com extensão específica para uso com a ferramenta Postman, permitindo assim que os testes sejam utilizados futuramente pela equipe responsável pelo Hawa sempre que necessário. Em conjunto com a suíte de testes automatizados, foi entregue um documento que deve ser utilizado como guia para uso dos testes automatizados com o Postman.

Por fim, sendo mais um resultado positivo do presente trabalho, foi entregue um documento baseado na documentação do Postman que pode ser utilizado como guia para a inclusão dos testes automatizados de integração implementados em futuras pipelines de validação. Também foi configurado e disponibilizado um repositório já preparado para execução dos testes automatizados implementados em uma pipeline de validação.

4.2. Resultados negativos

Poucos resultados negativos foram observados durante a execução do presente trabalho. Apenas 6 validações de um total de 63 demonstraram resultados preocupantes quanto ao estado da API REST, representando cerca de 10%. Ao analisar os resultados negativos obtidos em conjunto com a equipe responsável pelo Hawa, percebeu-se que os erros estavam relacionados à instabilidades no ambiente de teste e não ao software, fato que permitiu concluir que nenhum erro grave foi observado durante as validações.

Outro ponto negativo observado foi a percepção de que alguns dos endpoints da API REST do Hawa, devido ao contexto necessário para o uso dos mesmos, não são passíveis de automatização. Consequentemente esses endpoints precisam ser validados manualmente ou validados com auxílios de mocks.

Também foram percebidos alguns problemas com a documentação da API REST do software, pois como existem várias versões do produto, em alguns casos as documentações existentes não se mostraram integralmente corretas. Apesar de ser um ponto negativo, a oportunidade de melhorar as documentações do software surgiu como consequência.

6. Considerações finais

Este trabalho descreveu os conceitos relacionados e a importância dos testes automatizados para o contexto atual da indústria de software. Para visualizar na prática os benefícios da utilização dos testes automatizados, uma suíte de testes automatizados de integração foi planejada e implementada para uso na validação da API REST pública do software Hawa, sendo este carente de testes automatizados de integração e utilizado para a emissão de certificados digitais.

Espera-se que essa suíte de testes automatizados possa contribuir para a melhoria constante da qualidade do software citado, consequentemente tornando a emissão de certificados digitais mais confiável e segura. Documentos complementares guiando o uso futuro da suíte de testes automatizados foram escritos para que esse objetivo seja alcançado com maior facilidade pela equipe responsável pelo Hawa.

6.1. Trabalhos futuros

Oportunidades de melhorias foram percebidas analisando os pontos negativos do presente trabalho. Como primeira oportunidade de melhoria, é possível realizar correções e atualizações na documentação do software, garantindo assim um melhor entendimento por parte dos interessados no Hawa que tenham acesso aos documentos citados.

Também percebeu-se que alguns endpoints do Hawa, devido a características e contextos existentes, atualmente não são passíveis de validações automatizadas quando integrados a outros componentes. Desse fato surge a oportunidade de construir mocks que possam simular o funcionamento dos endpoints que não são passíveis de automatização.

Por fim, é possível configurar pipelines de validação automatizada para o Hawa, adicionando os próprios testes de integração automatizados implementados durante o presente trabalho como componente da estrutura de validação.

Referências

- FADEL, Aline Cristine; SILVEIRA, Henrique da Mota. Metodologias ágeis no contexto de desenvolvimento de software: XP, Scrum e Lean. Monografia do Curso de Mestrado FT - 027 - Gestão de Projetos e Qualidade da Faculdade de Tecnologia–UNICAMP, v. 98, p. 101, 2010.
- SEMEDO, Maria João Moreno. Ganhos de produtividade e de sucesso de Metodologias Ágeis VS Metodologias em Cascata no desenvolvimento de projectos de software. 2012.
- BERNARDO, Paulo Cheque; KON, Fabio. A importância dos Testes Automatizados. Engenharia de Software Magazine, v. 1, n. 3, p. 54-57, 2008.
- ICP - BRASIL. ITI - Instituto Nacional de Tecnologia da Informação, Brasília, 27 de Junho de 2017. Disponível em:<<https://www.iti.gov.br/icp-brasil>>. Acesso em: 15 de Outubro de 2019.
- DOS SANTOS SOARES, Michel. Comparação entre metodologias Ágeis e tradicionais para o desenvolvimento de software. INFOCOMP Journal of Computer Science, v. 3, n. 2, p. 8-13, 2004.
- DELAMARO, Marcio; JINO, Mario; MALDONADO, Jose. Introdução ao teste de software. Elsevier Brasil, 2017.
- RODRIGUEZ, Alex. Restful web services: The basics. IBM developer Works, v. 33, p. 18, 2008.