UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

BRUNO KEICHI FUJIWARA

ANÁLISE DE METODOLOGIAS E ESTRATÉGIAS DE TESTES DE API REST: UM ESTUDO DE CASO

São Carlos
2023

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

BRUNO KEICHI FUJIWARA

ANÁLISE DE METODOLOGIAS E ESTRATÉGIAS DE TESTES DE API REST: UM ESTUDO DE CASO

Monografia apresentada ao Departamento de Computação da Universidade Federal de São Carlos para obtenção do título de bacharel em Engenharia de Computação.

Orientação: Prof. Dr. Auri Marcelo Rizzo Vincenzi

São Carlos
2023

FUNDAÇÃO UNIVERSIDADE FEDERAL DE SÃO CARLOS DEPARTAMENTO DE COMPUTAÇÃO - DC/CCET
Rod. Washington Luís km 235 - SP-310, s/n - Bairro Monjolinho, São Carlos/SP, CEP 13565-905
Telefone: (16) 33518231 - http://www.ufscar.br

DP-TCC-FA nº 11/2023/DC/CCET

Graduação: Defesa Pública de Trabalho de Conclusão de Curso Folha Aprovação (GDP-TCC-FA)

FOLHA DE APROVAÇÃO

BRUNO KEICHI FUJIWARA

ANÁLISE DE METODOLOGIAS E ESTRATÉGIAS DE TESTES DE API REST: UM ESTUDO DE CASO

Trabalho de Conclusão de Curso

Universidade Federal de São Carlos – Campus São Carlos

São Carlos, 29 de agosto de 2023

ASSINATURAS E CIÊNCIAS

Cargo/Função	Nome Completo
Orientador	Auri Marcelo Rizzo Vincenzi
Membro da Banca 1	André Takeshi Endo
Membro da Banca 2	Delano Medeiros Beder

Documento assinado eletronicamente por **Andre Takeshi Endo**, **Professor(a) do Ensino Superior**, em 29/08/2023, às 15:23, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539 , de 8 de outubro de 201 5.

Documento assinado eletronicamente por **Delano Medeiros Beder**, **Professor(a) do Ensino Superior**, em 29/08/2023, às 15:26, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539 , de 8 de outubro de 201 5.

Documento assinado eletronicamente por **Auri Marcelo Rizzo Vincenzi**, **Docente**, em 01/09/2023, às 19:11, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539 , de 8 de outubro de 201 5.

A autenticidade deste documento pode ser conferida no site https://sei.ufscar.br/autenticaca o, informando o código verificador **1166959** e o código CRC **FE34FB34**.

DEDICATÓRIA

Dedico esse trabalho aos
meus pais, parentes e amigos.

AGRADECIMENTO

Agradeço primeiramente aos meus pais, Elisabeth e Dirceu, por sempre me proporcionarem as melhores condições, visando meu desenvolvimento pessoal e profissional, sem dúvidas foi essencial para que isso se tornasse possível. O carinho e incentivo de ambos sempre será motivo de gratidão e admiração em minha vida.

Ao meu orientador Auri por sua orientação e conselhos durante a realização deste trabalho.

À minha companheira Francielle, que sempre me apoiou e esteve ao meu lado, encorajando e dando forças para continuar seguindo em frente.

Aos amigos da graduação: Mineiro, Alan, Brunão, Rafa, Gabs, Jão e Zé por compartilharem momentos de alegria e dificuldade, sempre com muito companheirismo. Todos foram fundamentais também nessa jornada, e guardarei as lembranças eternamente em minha vida.

RESUMO

O presente trabalho tem como objetivo apresentar uma análise dos tipos de testes de API, suas características e a importância dos mesmos para garantir a qualidade de uma API. Além disso, será abordado o conceito de *Behavior-Driven Development* (BDD) e como ele pode ser aplicado na realização de testes de API. Serão apresentadas também algumas ferramentas que podem ser utilizadas na implementação do BDD em testes de API, assim como a comparação entre técnicas BDD e testes automatizados de API.

Os resultados indicam que os testes de API são fundamentais para garantir a qualidade de uma API, pois permitem verificar se a aplicação atende aos requisitos especificados e se está funcionando corretamente. A utilização do BDD na realização desses testes traz diversos benefícios, como a maior clareza na comunicação entre desenvolvedores e clientes, o aumento da eficiência na detecção de bugs e a facilidade de manutenção dos testes.

Além disso, a comparação entre as técnicas BDD e testes automatizados de API demonstrou que ambas são importantes e complementares, podendo ser utilizadas em conjunto para obter melhores resultados.

Palavras-chave: API. Testes de API. Behavior-Driven Development. BDD. Testes automatizados.

ABSTRACT

The present work aims to present an analysis of API testing types, their characteristics, and the importance of these tests to ensure API quality. In addition, the concept of Behavior-Driven Development (BDD) and how it can be applied in API testing will be addressed. Some tools that can be used to implement BDD in API testing will also be presented, as well as a comparison between BDD techniques and automated API testing.

The results indicate that API testing is essential to ensure API quality, as it allows verifying if the application meets the specified requirements and is functioning correctly. The use of BDD in performing these tests brings several benefits, such as greater clarity in communication between developers and clients, increased efficiency in bug detection, and ease of test maintenance.

Furthermore, the comparison between BDD techniques and automated API testing demonstrated that both are important, complementary and can be used together to obtain better results.

Keywords: API. API Testing. Behavior-Driven Development. BDD. Automated Testing.

SUMÁRIO

1 INTRODUÇÃO

As APIs (*Application Programming Interfaces*) são cada vez mais utilizadas nos últimos anos para a comunicação entre aplicações e sistemas distribuídos. A facilidade de integração, o baixo custo e a escalabilidade fazem das APIs uma das tecnologias mais usadas para a criação de aplicações modernas e robustas (MAILGUN, 2021). No entanto, com a utilização em larga escala dessas interfaces, é necessário garantir que elas estejam funcionando de forma correta e eficiente. Por isso, os testes de API são tão importantes.

Porém, ao mesmo tempo, existe uma psicologia por trás dos testes onde a maioria dos programadores dizem que “o teste é o processo de demostrar que os erros não estão presentes” ou “o teste é o processo de estabelecer a confiança de que um programa faz o que é suposto a fazer”, tais falsas definições são uma das principais causas dos testes serem ruins (MYERS, 2011).

Os testes de API são uma prática essencial para garantir que as APIs estejam funcionando de acordo com o esperado, ou seja, que elas estejam fornecendo as informações corretas e que estejam operando de maneira eficiente. Além disso, os testes de API também são responsáveis por garantir que as mudanças feitas nas APIs não impactem negativamente em outras partes da aplicação ou em outros sistemas que as utilizem.

Uma API acaba expondo dados e serviços comerciais através de uma interface. Os desenvolvedores utilizam essa interface para criar aplicações que dependem da API. Esta aplicação invoca a API com uma grande quantidade de dados de entrada O tráfego da API varia com a utilização da aplicação. A API deve ser capaz de lidar com o tráfego em diferentes cargas. Com todas essas combinações e cenários, a importância do teste de API aumenta muito (BRAJESH, 2017).

Assim, neste trabalho, serão abordados diferentes tipos de testes de API para mostrar a importância e relevância de cada uma dentro do cenário de desenvolvimento, apresentar também as ferramentas utilizadas para a implementação desses testes e como a técnica BDD (CRISPIN, 2009) pode ser aplicada nos testes de API para garantir a eficiência e a integridade das APIs.

O restante deste trabalho está dividido da seguinte forma: cap. 2 contemplando os conceitos sobre os testes e as ferramentas utilizadas, cap. 3 contemplando o desenvolvimento e o resultado dos testes, cap. 4 e 5 apresentando sobre as lições aprendidas e conclusão.

2 REVISÃO BIBLIOGRÁFICA

2.1 API

A API (*Application Programming Interface*) é uma interface que obedece a um conjunto de padrões e protocolos que permitem que diferentes softwares se comuniquem entre si. Ela define a maneira como as solicitações e respostas devem ser formatadas, de modo que os sistemas compartilhem dados de forma padronizada.

De um modo geral, uma API expõe um conjunto de dados e funções para facilitar as interações entre programas de computador e permitir que eles troquem informações (MASSE, 2011). Assim, qualquer um que possua conhecimento em relação a documentação da API pode estar consumindo-a, dispensando o desenvolvimento de uma aplicação que possua a mesma função.

2.1.1 API REST

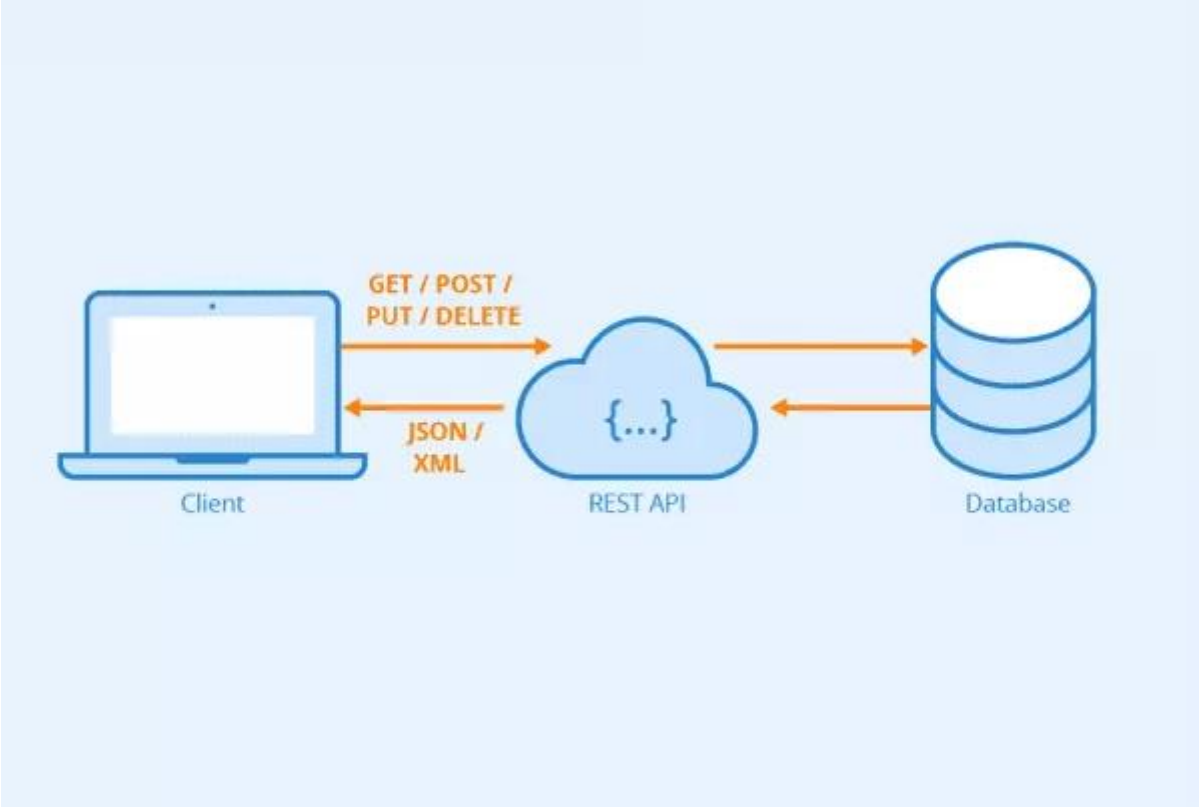
API REST é um estilo arquitetural de projeto de *software* que utiliza o protocolo HTTP para permitir a comunicação entre sistemas distribuídos. O termo REST significa *Representational State Transfer*, ou seja, Transferência de Estado Representacional. A ideia central do REST é o uso dos recursos disponíveis na *Web*, identificados por meio de URIs (*Uniform Resource Identifiers*), e a utilização dos métodos HTTP para manipulação desses recursos.

De acordo com Fielding (2000), um dos principais autores e criadores do conceito de REST, uma API REST deve atender a alguns princípios, como: utilização do protocolo HTTP; utilização de URIs para identificação dos recursos; utilização de verbos HTTP para operações básicas de CRUD (*Create, Read, Update, Delete*); uso de hipermídia como mecanismo de estado da aplicação.

Em resumo, a API REST é uma arquitetura de software que permite a comunicação entre sistemas distribuídos, utilizando o protocolo HTTP e seguindo alguns princípios como a utilização de URIs e verbos HTTP para manipulação de recursos. A adoção de uma API REST

pode trazer diversos benefícios para o desenvolvimento de aplicações distribuídas, como a facilidade de integração e a escalabilidade. Esta representação pode ser observada na Figura 1.

FIGURA 1 – REPRESENTAÇÃO DE ESTADO DA API



Fonte: Definição da *API REST*. Disponível em: https://www.astera.com/pt/tipo/blog/defini%C3%A7%C3%A3o-de-api/. Acesso em: 20 de jun. 2023

2.1.2 Protocolo HTTP

O protocolo HTTP define um conjunto de métodos de requisição responsáveis por indicar a ação a ser executada para um dado recurso. Embora esses métodos possam ser descritos como substantivos, eles também são comumente referenciados como Verbos HTTP (MOZILLA). Entre eles, são:

- GET: solicita a representação de um recurso específico e as requisições utilizando esse método devem retornar apenas dados.
- HEAD: solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta.

- POST: utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.
- PUT: substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.
- DELETE: remove um recurso específico.
- CONNECT: estabelece um túnel para o servidor identificado pelo recurso de destino.
- OPTIONS: utilizado para descrever as opções de comunicação com o recurso de destino.
- PATCH: utilizado para aplicar modificações parciais em um recurso.

2.2 Testes de API

Testes de API são procedimentos que envolvem a verificação do comportamento, funcionalidade, desempenho e segurança de uma API, com o objetivo de garantir que a API está atendendo aos requisitos especificados e que está fornecendo os resultados esperados.

O teste de API é um dos aspectos mais difíceis do processo de teste de software e teste de controle de qualidade, pois enquanto o desenvolvedor tende a testar apenas os recursos que estão trabalhando, os testadores são responsáveis por avaliar tanto a funcionalidade individual quanto uma série de recursos para ver como elas interagem do início ao fim. (Thalayasingam, 2021).

2.2.1 Tipos de Testes de API

Existem diversos tipos de testes que podem ser aplicados em APIs (NORDICAPIS), dentre eles:

2.2.1.1 Teste de Integração

Teste de integração de API é uma técnica de teste de *software* que verifica a comunicação e a interoperabilidade entre diferentes componentes de um sistema por meio das APIs. É um tipo de teste de *software* que é executado após a conclusão dos testes de unidade e antes dos testes de sistema. Basicamente, o objetivo do teste de integração, como o nome

sugere, é testar se muitos módulos desenvolvidos separadamente funcionam juntos conforme o esperado (FOWLER, 2018).

O teste de integração é conhecido como o segundo nível do processo de teste de software, seguindo o teste de unidade. O teste envolve a verificação de componentes individuais ou unidades de um projeto de software para expor defeitos e problemas para verificar se eles funcionam juntos conforme projetado (Terra, 2023).

2.2.1.2 Teste de Validação

Teste de validação de API é um tipo de teste de software que tem como objetivo verificar se a API está retornando os resultados esperados de acordo com a especificação definida para ela. Esse tipo de teste é utilizado para garantir que as regras de negócio e funcionalidades implementadas na API estão de acordo com os requisitos do usuário e com as expectativas do sistema em que a API será integrada.

O teste de validação de API pode incluir a verificação dos parâmetros de entrada e saída, o formato dos dados retornados, a validação de códigos de status HTTP e a verificação de possíveis erros e exceções. É importante que esses testes sejam automatizados para garantir a eficiência e confiabilidade dos resultados. Diferentemente do teste de integração que verifica se todos os módulos funcionam de forma conjunta, o teste de validação verifica o módulo de forma individual.

De acordo com API Mike, o teste de validação de API é uma etapa importante no processo de desenvolvimento de software, pois ajuda a garantir que uma API esteja funcionando corretamente e que atenda às necessidades de seus usuários. Além disso, o teste de validação de API pode ajudar a detectar possíveis problemas antes que eles afetem os usuários finais e a evitar retrabalho e custos adicionais.

2.2.1.3 Teste de Segurança

Teste de segurança de API é uma prática que visa avaliar a segurança de uma API relacionado à vulnerabilidade e ataques cibernéticos, com o objetivo de identificar brechas de segurança, verificar a proteção contra-ataques de injeção, validar a autenticação e autorização

corretas, avaliar a proteção dos dados transmitidos e garantir a conformidade com padrões de segurança.

A falta de testes de segurança pode prejudicar em diversos aspectos, na qual a coisa mais importante a considerar é a perda real de dados ou danos aos dados que podem causar todos os tipos de problemas para sua organização. A recuperação de dados é um processo caro e sujeito a erros que custará mais do que tempo e dinheiro (SMARTBEAR).

Segundo o OWASP (*Open Web Application Security Project*), as principais categorias de vulnerabilidades de segurança de API incluem:

- Autenticação e autorização inadequadas;
- Exposição de informações sensíveis;
- Manipulação inadequada de entradas e saídas;
- Falhas na lógica de negócios;
- Vulnerabilidades de servidor e infraestrutura;
- Controles de segurança ausentes ou ineficazes.

Para realizar testes de segurança em APIs, existem várias ferramentas disponíveis, como OWASP ZAP, Burp Suite[1], SoapUI[2] entre outras. É importante ressaltar que o teste de segurança não deve ser um processo isolado, mas sim integrado ao ciclo de vida de desenvolvimento de *software*, desde o planejamento até a manutenção.

2.2.1.4 Teste de Performance

Testes de performance de API são realizados para avaliar o desempenho e a escalabilidade de uma API. Esses testes são importantes para garantir que o serviço não tenha problema de desempenho, a manter o tempo de resposta dentro de limites aceitáveis e ajudar a entender como a API irá lidar com os diferentes tipos de tráfego. Os testes de performance de API geralmente envolvem a simulação de muitas solicitações, com a finalidade de identificar gargalos de desempenho, falhas de escalabilidade e possíveis problemas de segurança.

[1] https://portswigger.net/burp
[2] https://www.soapui.org/docs/security-testing/getting-started/

Uma das principais ferramentas para testes de performance de API é o JMeter[3], que permite simular muitos usuários, solicitações e cargas de trabalho para avaliar a capacidade de resposta do sistema em diferentes condições.

Além do JMeter, outras ferramentas podem ser usadas para testes de performance de API, como o Apache Bench[4], o LoadRunner[5] e o Gatling[6]. No entanto, a escolha da ferramenta dependerá das necessidades específicas do projeto e dos recursos disponíveis para a equipe de teste.

Em resumo, os testes de performance de API são importantes para garantir que um serviço de API possa lidar com uma carga de trabalho esperada e manter o tempo de resposta dentro de limites aceitáveis. O uso de ferramentas como o JMeter permite simular uma grande variedade de cenários de uso e identificar possíveis gargalos de desempenho.

2.2.1.5 Teste Funcional

Teste funcional de API é uma técnica de teste que verifica a funcionalidade correta de um sistema por meio da execução de testes nas APIs. Esses testes são realizados para verificar se as APIs estão funcionando conforme o esperado, se estão cumprindo suas funções e se os dados de entrada e saída estão corretos.

Os testes funcionais de API podem ser automatizados usando frameworks e ferramentas específicas, como o Postman e o RestAssured[7]. Eles também podem ser integrados a metodologias de desenvolvimento ágil, como o BDD (*Behavior Driven Development*) (ELLIOT, 2019) e o TDD (*Test Driven Development*) (ELLIOT, 2019), permitindo que os testes sejam escritos antes mesmo do código ser desenvolvido, tornando o processo de desenvolvimento mais eficiente.

Em resumo, os testes funcionais de API são fundamentais para garantir que as APIs estejam funcionando corretamente, atendendo às expectativas do usuário e cumprindo suas funções de negócio. A automação desses testes pode ser feita usando ferramentas e *frameworks*

[3] https://jmeter.apache.org/
[4] https://httpd.apache.org/docs/2.4/programs/ab.html
[5] https://www.microfocus.com/pt-br/products/loadrunner-professional/overview
[6] https://gatling.io/
[7] https://rest-assured.io/

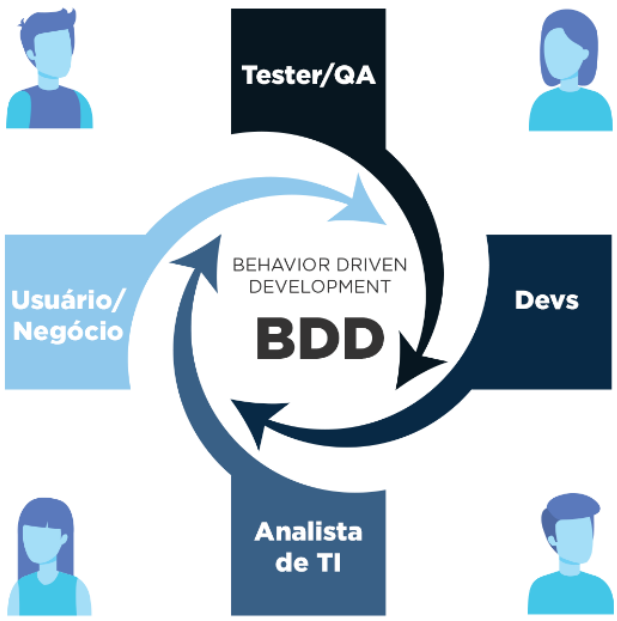
específicos, e sua integração com metodologias ágeis pode tornar o processo de desenvolvimento mais eficiente e com maior qualidade.

2.3 *Behavior-Driven Development* (BDD)

Behavior-Driven Development (BDD) é um conjunto de práticas de engenharia de software projetado para ajudar as equipes a criar e entregar o software com alta qualidade com rapidez. A técnica se baseia em práticas ágeis e enxutas, fornecendo uma linguagem comum baseada em frases simples e estruturadas, que facilitam a comunicação entre os membros do time do projeto e partes interessadas do negócio (SMART, 2014).

Basicamente é uma técnica que visa aprimorar a colaboração entre desenvolvedores, testadores e partes interessadas de um projeto, promovendo uma compreensão dos requisitos do sistema. O BDD se concentra na descrição de cenários de testes em uma linguagem acessível a todas as partes envolvidas.

FIGURA 2 – REPRESENTAÇÃO DA TÉCNICA BDD



Fonte: O que é BDD? Disponível em: https://www.eteg.com.br/blog/o-que-e-bdd/. Acesso em: 20 de jun. 2023.

2.3.1 Gherkin

A linguagem Gherkin é uma linguagem de domínio específico (DSL - Domain-Specific Language) que é utilizada dentro do contexto de BDD. Segundo SMART (2014), a linguagem é usada para descrever o comportamento do software em um formato de texto legível, permitindo especificar os requisitos e cenários de teste de forma clara e concisa.

Estes cenários de teste são escritos em uma linguagem acessível a todas as partes interessadas, incluindo desenvolvedores, testadores e stakeholders de negócios, conforme ilustrado na Figura 3.

FIGURA 3 – EXEMPLO DE CENÁRIO DE LOGIN

```
# coding: utf-8
# language: pt

Funcionalidade: Login
  Eu como usuário, desejo realizar o login
  Para que eu possa ter acesso as principais funcionalidades da aplicação

  @done @ios @android @CI @smoke_test
  Cenário: O usuário pode realizar seu login na aplicação
    Dado que estou na tela de login
    Quando realizar o login com usuário válido
    Então o login é realizado com sucesso
```

Fonte: Cucumber: truques e dicas. Disponível em: https://imasters.com.br/desenvolvimento/cucumber-truques-e-dicas . Acesso em: 20 de jun. 2023.

2.4 Visual Studio 2022

O Visual Studio 2022 é a versão mais recente do ambiente de desenvolvimento integrado (IDE) da Microsoft, lançado em novembro de 2021. Ele é utilizado por desenvolvedores para criar aplicativos para diversas plataformas, como Windows, Android, iOS, *web* e nuvem. O Visual Studio 2022 possui diversas ferramentas que auxiliam os desenvolvedores no processo de criação de aplicativos, incluindo depuração, testes, gerenciamento de código e integração com sistemas de controle de versão. Além disso, ele oferece suporte a diversas linguagens de programação, incluindo C#, C++, JavaScript, Python, entre outras.

De acordo com a Microsoft (2022), o Visual Studio 2022 traz diversas melhorias em relação às versões anteriores, como maior desempenho e estabilidade, novos recursos de

produtividade, melhorias na interface de usuário e suporte a novas tecnologias. Além disso, ele também oferece ferramentas para desenvolvimento de aplicativos para a nuvem, como o Azure, bem como suporte a ferramentas de inteligência artificial e aprendizado de máquina.

2.5 Apache JMeter

O Apache JMeter é uma ferramenta de teste de desempenho *open-source* e 100% Java, desenvolvida pela Apache Software Foundation (APACHE). Ela é usada para testar a capacidade de carga de sistemas web, incluindo APIs, simulando vários usuários acessando a aplicação simultaneamente. O JMeter é capaz de gerar e enviar solicitações HTTP e HTTPS, e medir o tempo de resposta delas, permitindo que o desempenho do sistema seja avaliado e ajustado de acordo com as necessidades, de acordo com o próprio projeto Apache JMeter.

O JMeter oferece diversas funcionalidades, incluindo a possibilidade de criar planos de teste personalizados, configurar regras de extração de dados de respostas HTTP, e integrar com outras ferramentas de teste e automação. Ele é amplamente utilizado por equipes de desenvolvimento e testes para avaliar o desempenho de sistemas em diferentes cenários de carga.

2.6 Postman

Postman é uma ferramenta de colaboração para o desenvolvimento de APIs, que permite aos desenvolvedores criar, testar, documentar e compartilhar APIs em um único ambiente (POSTMAN, 2023). Com a ferramenta Postman, os desenvolvedores podem testar seus *endpoints* de API com facilidade, automatizar testes, monitorar o desempenho da API e colaborar com outros membros da equipe de desenvolvimento. Além disso, o Postman oferece recursos avançados, como testes de integração, monitoramento de desempenho, autenticação e gerenciamento de coleções.

Segundo o site oficial do Postman (2023), "mais de 17 milhões de desenvolvedores em todo o mundo e mais de 500.000 empresas confiam no Postman como sua ferramenta de colaboração para desenvolvimento de API".

O Postman é compatível com diferentes sistemas operacionais, como Windows, macOS e Linux, e possui uma interface fácil de usar que permite aos desenvolvedores criar e testar APIs de maneira eficiente. Além disso, a ferramenta oferece recursos de documentação de API que ajudam a manter a documentação atualizada e a garantir que as APIs sejam utilizadas de acordo com as especificações.

2.7 SpecFlow

SpecFlow é uma ferramenta de teste de aceitação baseada em BDD (*Behavior Driven Development*) que permite a colaboração entre desenvolvedores, testadores e outras partes interessadas no processo de desenvolvimento de *software*. Ela permite escrever especificações de recursos em linguagem natural, como o Gherkin, que pode ser facilmente entendida por pessoas de negócios e desenvolvedores.

De acordo com a documentação oficial do SpecFlow (2023):

"SpecFlow é um framework de teste de aceitação .NET que suporta *Behavior Driven Development* (BDD). Ele permite que os usuários escrevam cenários em uma linguagem natural que é facilmente compreendida por todos os membros da equipe. Esses cenários podem ser executados como testes automatizados, permitindo que as equipes verifiquem se o *software* está funcionando corretamente e se atende às necessidades do cliente."

O SpecFlow permite que os testes de aceitação escritos em linguagem natural sejam automatizados e integrados com ferramentas de teste como o NUnit[8] e o Visual Studio Test Explorer[9].

2.8 OWASP

OWASP (*Open Web Application Security Project*) é uma comunidade de profissionais de segurança cibernética que tem como objetivo melhorar a segurança de software em todo o mundo. A comunidade é conhecida por suas listas de 10 principais vulnerabilidades de

[8] https://nunit.org/
[9] https://docs.telerik.com/teststudio/automated-tests/vs-plugin/vs-test-explorer

segurança em aplicativos da web, que são amplamente utilizadas pela indústria de segurança cibernética como um guia para avaliar a segurança de aplicativos da web.

Segundo a OWASP (2022), a organização "foca em melhorar a segurança de software". A missão é tornar o software seguro. O OWASP é um fórum aberto para qualquer pessoa interessada em aprender mais sobre segurança de software. As atividades OWASP incluem a publicação de artigos, metodologias, documentação, ferramentas e tecnologias relacionadas à segurança de software. O OWASP também realiza conferências, treinamentos e trabalha para definir padrões de segurança de software. O OWASP é um esforço de colaboração aberto liderado por voluntários."

A organização fornece uma série de recursos gratuitos, incluindo guias de teste de segurança, ferramentas de segurança de código aberto, e uma comunidade de profissionais de segurança cibernética. Seu trabalho é considerado um referencial para o desenvolvimento seguro de software e é amplamente utilizado pela indústria de segurança cibernética em todo o mundo.

3 DESENVOLVIMENTO DOS TESTES

3.1 VISÃO GERAL

Neste item são apresentados os detalhes do desenvolvimento dos testes de API, mostrando os tipos de testes realizados e a utilização da técnica (BDD).

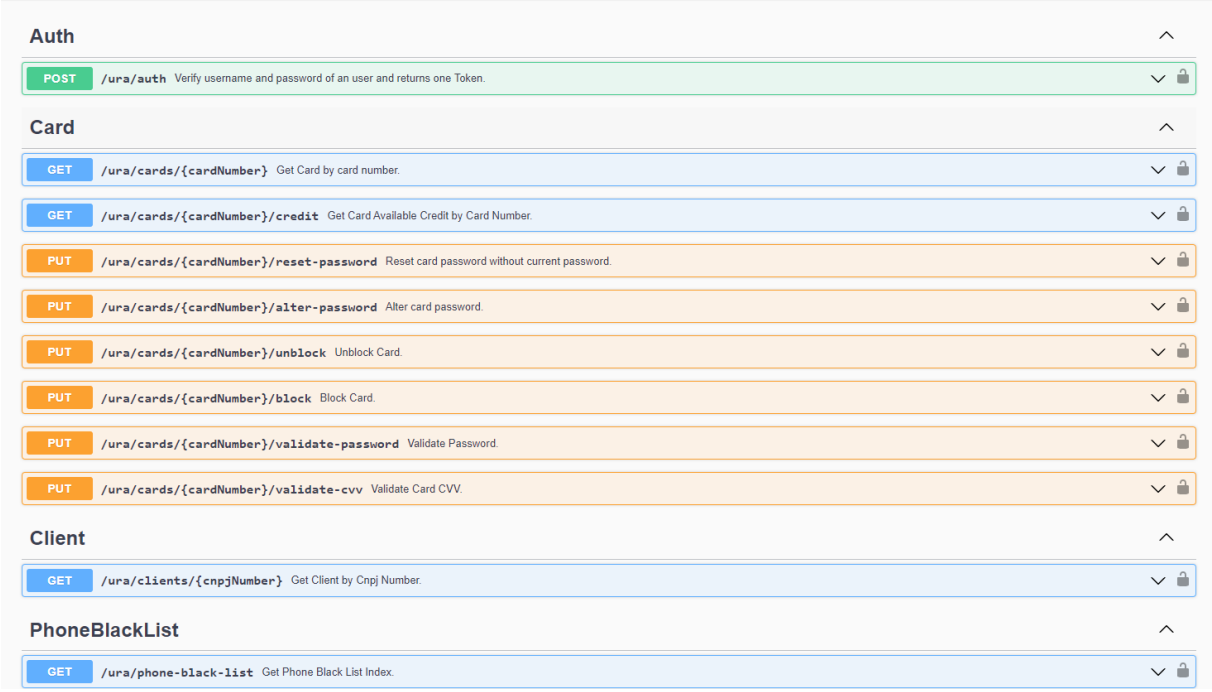
A API escolhida para este trabalho foi a API URA da empresa Up Brasil, uma empresa multinacional de benefícios corporativos, detentora dos direitos e responsável pela API, e o uso da API foi autorizado por eles. Esta aplicação é utilizada pelas soluções multicanais para atendimento de usuários das soluções Up Brasil. Soluções como bloqueio e desbloqueio de cartão, *reset* e troca de senha do cartão podem ser feitos via URA (Unidade de Resposta Audível).

A URA é uma tecnologia que permite que os clientes interajam com o sistema de atendimento das empresas via ligação telefônica. Esta interação pode ser feita por reconhecimento de voz ou por meio das teclas do telefone através da tecnologia DTFM (*Dual Tone Multi Frequency*), onde as diferentes frequências sonoras geradas pelo som da tecla digitada são capturada pelo sistema, e entende o que deve ser feito. Os clientes normalmente respondem a áudios pré-gravados ou a mensagens de voz geradas dinamicamente por mecanismos de síntese de voz.

3.2 IMPLEMENTAÇÃO DOS TESTES

Para a realização dos testes na API URA, foram criados diferentes cenários e implementados diversos tipos de teste para cada *endpoint*, onde foram verificados cenários de falha e sucesso. Todos os testes na API seguiram e respeitaram as regras de negócio para cada *endpoint*. A Figura 4 mostra os *endpoints* da API URA conforme documentado pela ferramenta Swagger.

FIGURA 4 – DOCUMENTAÇÃO SWAGGER



Fonte: próprio autor.

Na Figura 4 é ilustrada a documentação Swagger da API URA na qual são elencadas as requisições e rotas disponíveis. Os métodos utilizados nessa API são GET, POST E PUT.

3.2.1 Teste de Validação

No desenvolvimento do teste de validação, utilizamos a ferramenta Postman, a qual é suficientemente completa e eficaz. A ferramenta possui uma aplicação nativa para o sistema operacional Windows 10, o que faz com o que o processo de instalação do Postman seja fácil e intuitivo.

No Postman é possível configurar a requisição adicionando scripts na linguagem JavaScript, para ser executado após o recebimento da resposta da requisição e antes do envio da requisição. Nesse teste, em específico, o script será executado após o recebimento da resposta.

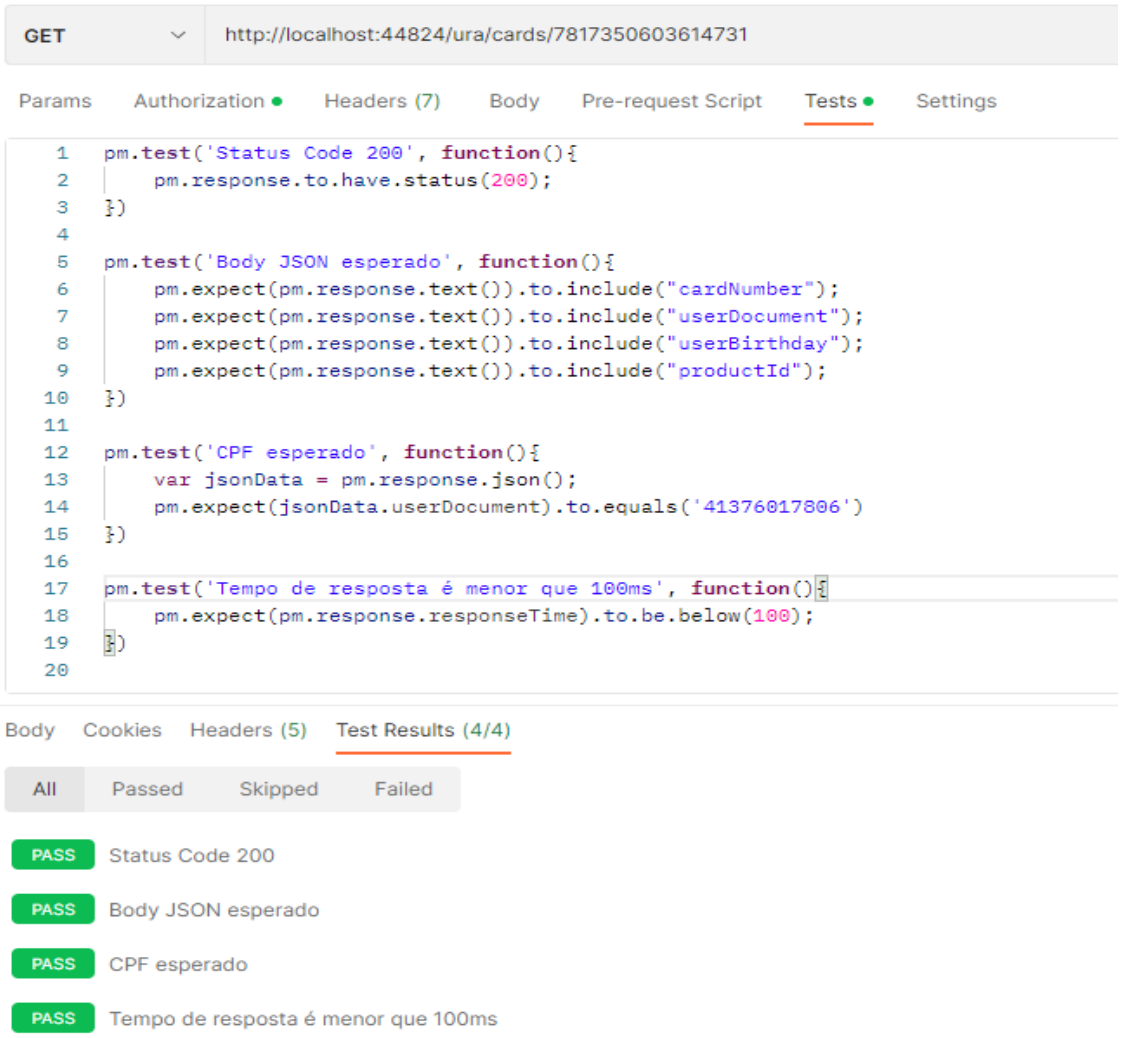
Nesses casos de testes, foi configurada uma requisição no Postman e foi codificado um script de teste para validar a resposta recebida ao enviar a requisição, com implementação de parâmetros válidos e inválidos para validar o funcionamento do *endpoint* em diversas

situações. O *endpoint*, ao receber os parâmetros válidos, o *endpoint* retornou o resultado esperado; e ao receber os parâmetros inválidos, forneceu a resposta e *feedback* compatíveis.

O *endpoint* escolhido para configurar uma requisição foi o *GetCard*, contendo o parâmetro "*cardNumber*". Esse *endpoint* é responsável por retornar, em um objeto JSON, os dados do cartão de um usuário, como o número do cartão, o CPF do usuário, o tipo de produto do cartão (Vale Alimentação, Vale Refeição etc.), o crédito disponível no cartão e entre outros dados.

Na Figura 5 é ilustrado um *script* de teste para validar se o status retornado será 200, se o corpo do objeto JSON terá as propriedades esperadas, se o CPF retornado será o esperado e se o tempo de resposta da requisição será menor que 100ms, isso para um determinado número de cartão como parâmetro.

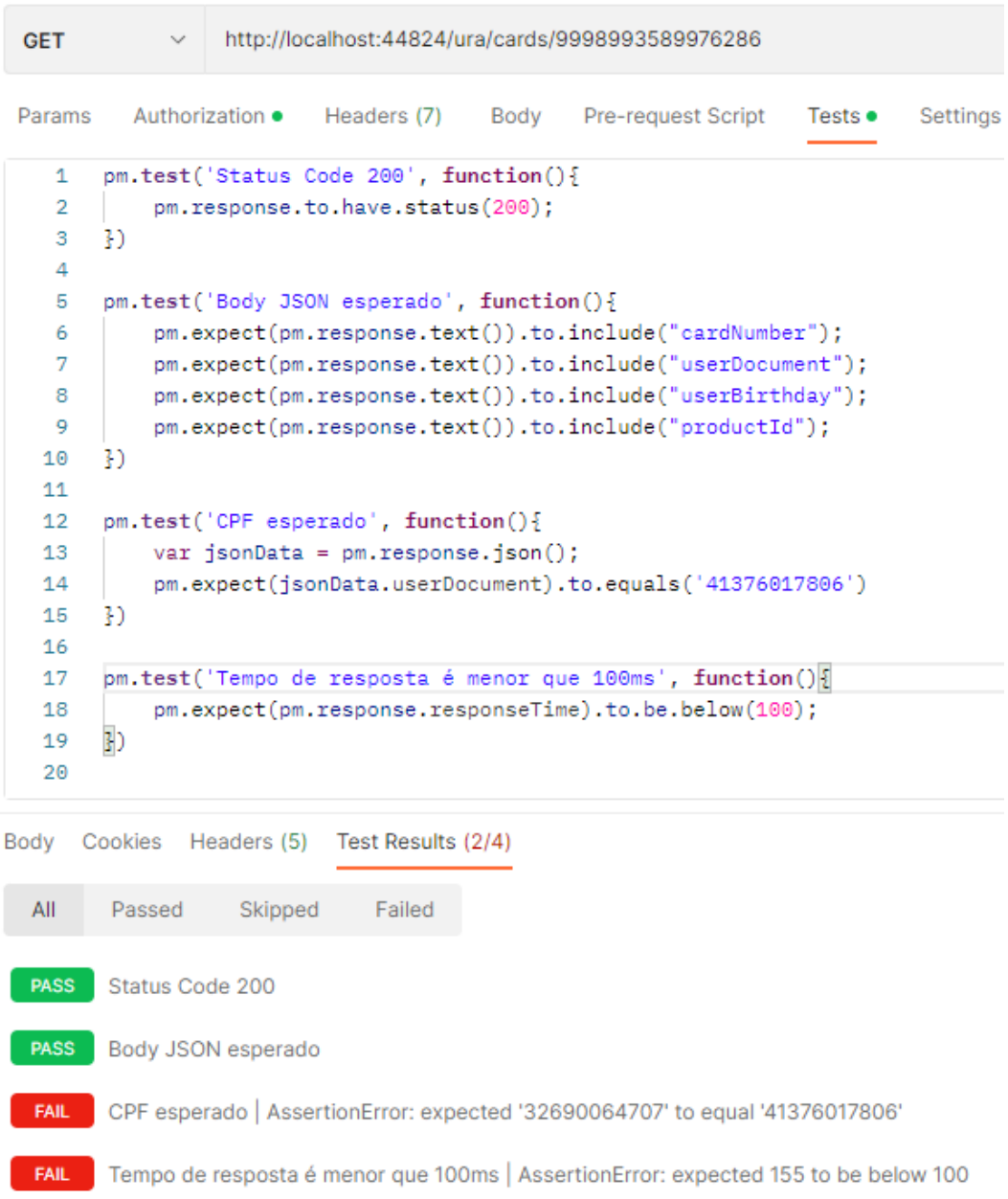
FIGURA 5 – TESTE DE VALIDAÇÃO 1



Fonte: próprio autor.

Utilizando o mesmo *script* de validação com um parâmetro diferente, ou seja, um outro número de cartão, o resultado das validações diferiu comparado com o resultado anterior, conforme ilustrado na Figura 6.

FIGURA 6 – TESTE DE VALIDAÇÃO 2



Fonte: próprio autor.

Estes testes de validação foram importantes para validar as entradas e saídas da API, com o objetivo de identificar, por exemplo, possíveis erros e problemas antes de ser implantado no ambiente de produção; e na validação da conformidade com os padrões e requisitos

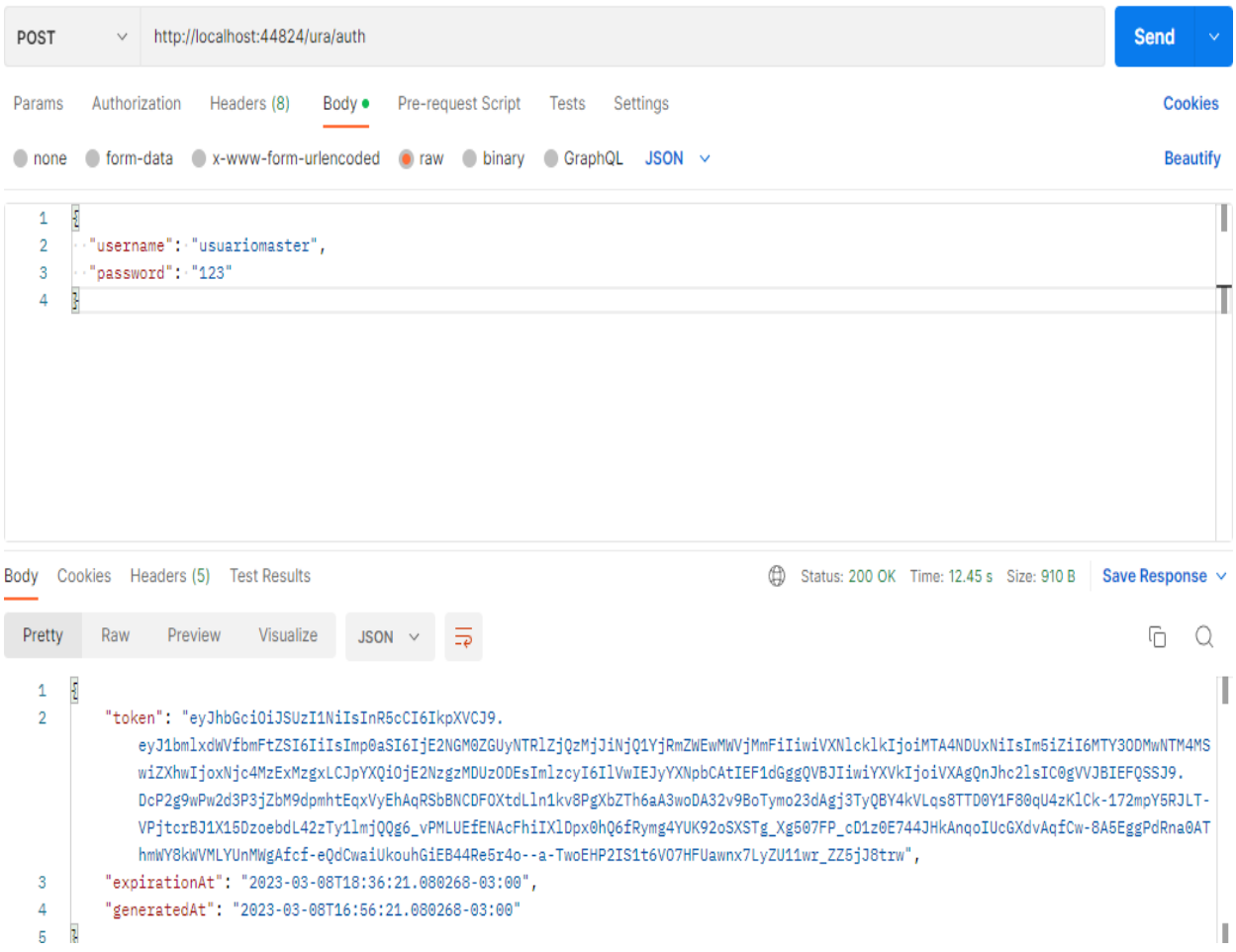
definidos. Por outro lado, este teste exige um esforço significativo na criação de cenários e pode deixar lacunas na cobertura de testes em outros aspectos do sistema, por validar somente a entrada e saída.

A utilização do Postman para este teste foi devida à sua interface simples, permitindo a criação de testes de forma rápida. A ferramenta também oferece ótimos recursos para a depuração e execução dos testes e, particularmente, isso é bem útil durante os testes.

3.2.2 Teste Funcional

Para o desenvolvimento dos testes funcionais da API, também foi utilizada a ferramenta Postman. Com o serviço em execução (localmente), iniciou-se os testes pelo *endpoint* de autenticação via POST, conforme ilustrado na Figura 7.

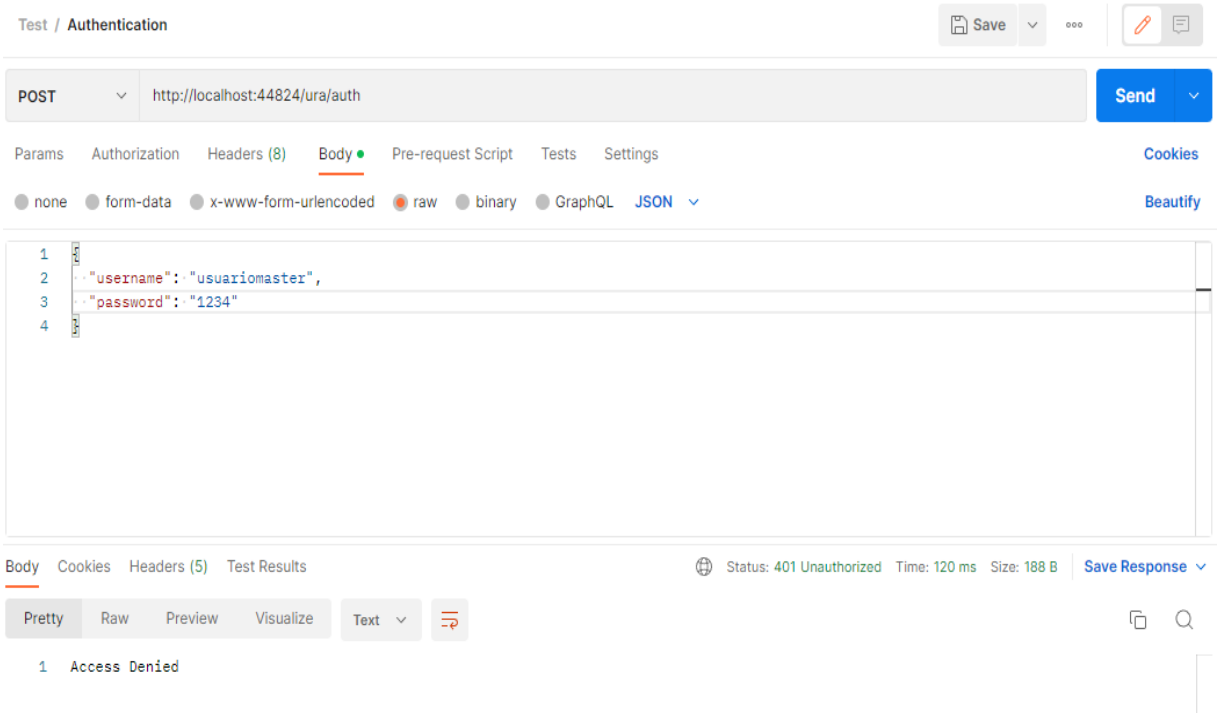
FIGURA 7 – TESTE FUNCIONAL 1



Fonte: próprio autor.

A Autenticação Bearer (*Bearer Authentication*) ou conhecido também como *Token Authentication* é uma *Schema* para autenticação *HTTP*. Como o usuário utilizado em questão existe na base de dados, o *endpoint auth* retornou um token que representa uma autorização do *server* emitida para um cliente, que foi utilizado, posteriormente, para permitir a utilização de outros *endpoints*. Notou-se também que o status da operação foi 200 (OK) e o tempo que demorou para obter o resultado foi de 12,45s, diferentemente quando passamos como parâmetro usuários que não constam na base, ilustrado na Figura 8.

FIGURA 8 – TESTE FUNCIONAL 2

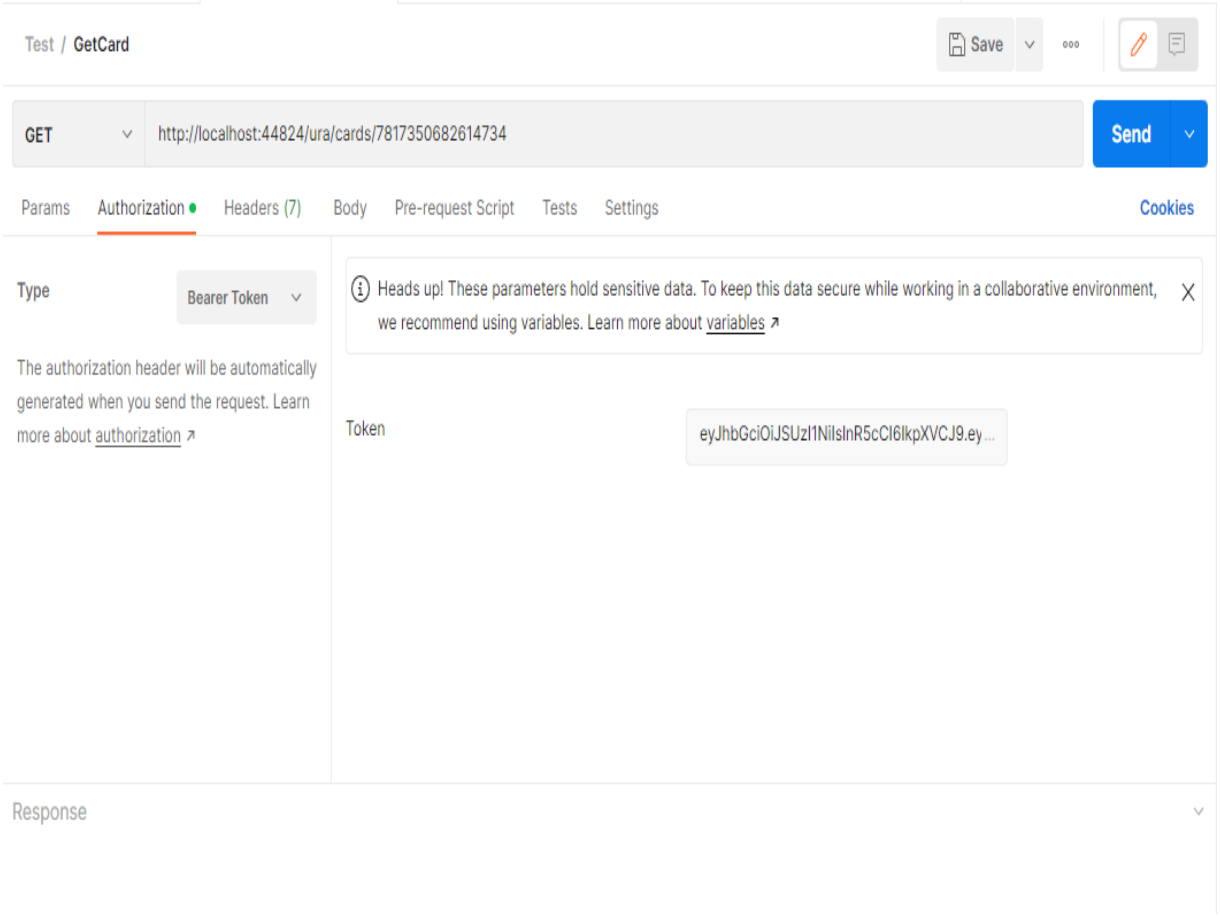


Fonte: próprio autor.

O mesmo usuário com uma senha diferente não consta na base de dados, logo o *endpoint* de autorização nega o acesso ao serviço, impossibilitando o usuário de utilizar a API, como pode ser observado pelo erro 401 (*Unauthorized*).

O próximo *endpoint* que foi testado foi o *GetCards*, que retornou os dados de um cartão via GET, possuindo como parâmetro o número do cartão. Antes de iniciar essa chamada, foi utilizado o *token* obtido pelo endpoint Auth, como mostrado na Figura 9.

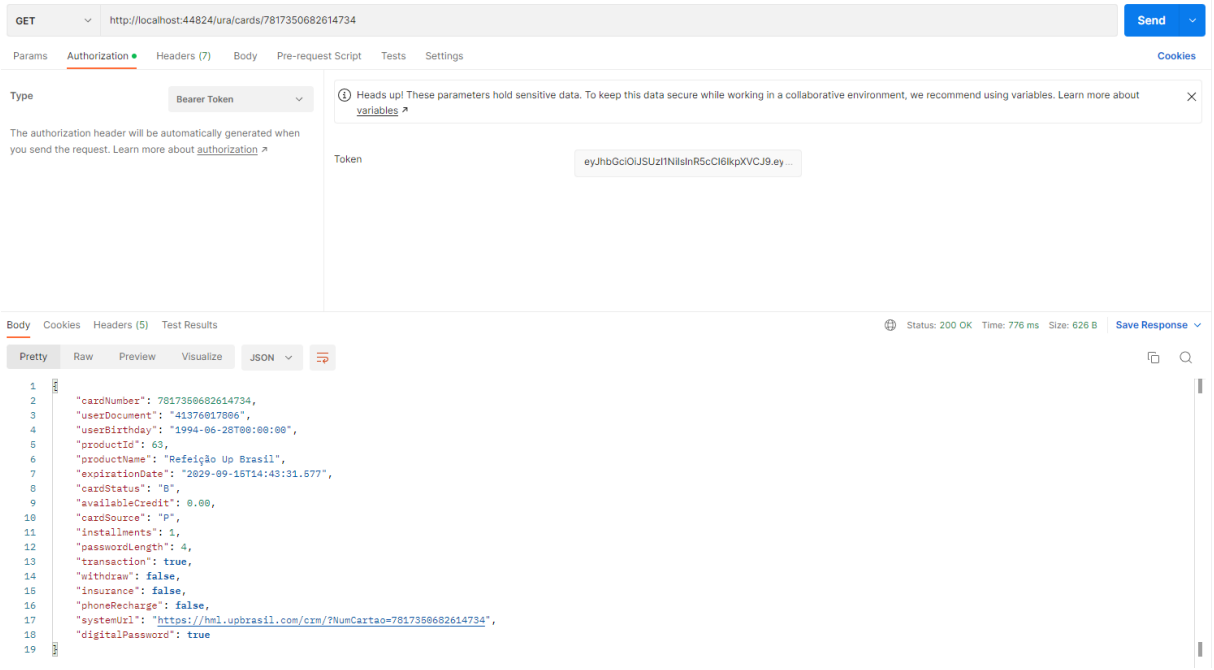
FIGURA 9 – BEARER TOKEN



Fonte: próprio autor.

O *token* obtido foi utilizado na aba *Authorization* do Postman, com o *type* escolhido como *Bearer Token*. Após preenchido, a requisição foi enviada, juntamente com um número de cartão válido como parâmetro, e retornou o seguinte objeto em JSON, conforme ilustrado na Figura 10.

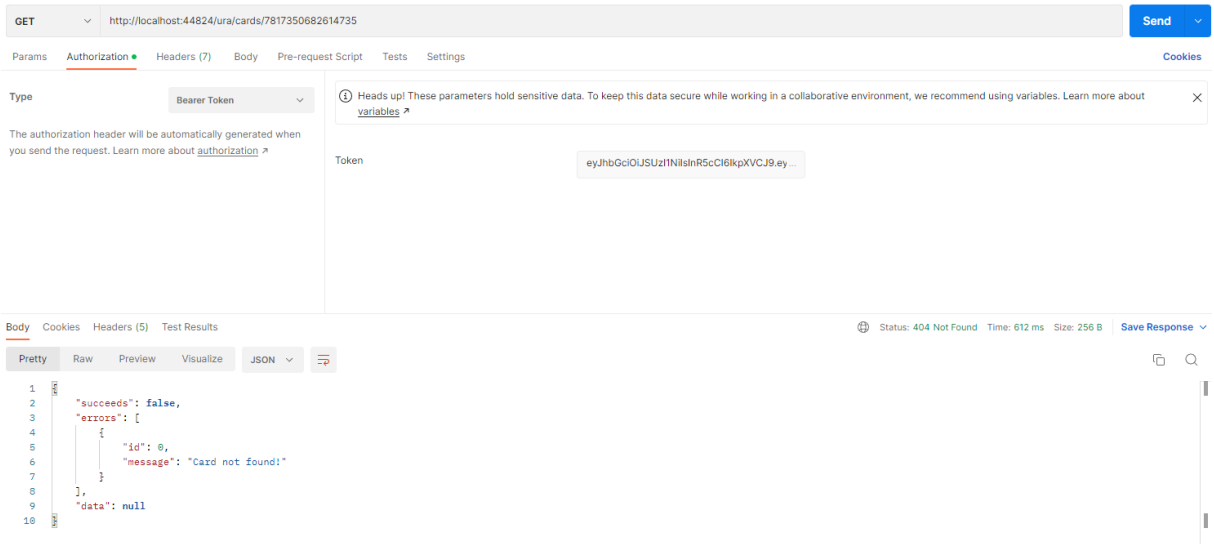
FIGURA 10 – TESTE FUNCIONAL 3



Fonte: próprio autor.

O objeto JSON retornado possui o número do cartão, documento do usuário responsável pelo cartão, tipo do produto do cartão (Alimentação, Refeição etc.), crédito disponível, tamanho da senha e entre outros dados, cumprindo com a função proposta do *endpoint*. O retorno é diferente quando foi utilizado um número de cartão não existente, conforme mostrado na Figura 11.

Figura 11 – TESTE FUNCIONAL 4

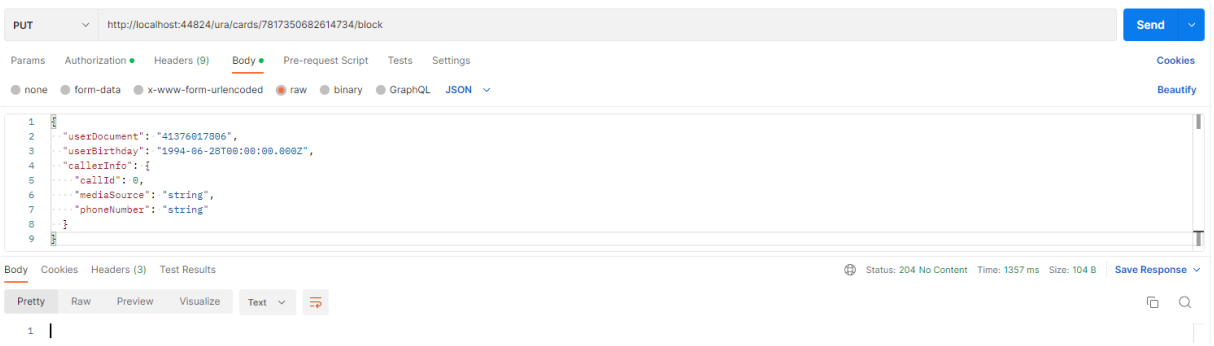


Fonte: próprio autor.

O objeto retornado informa que o cartão não foi encontrado e, consequentemente, não trouxe nenhum dado do cartão no corpo do objeto. Notou-se que o status da operação foi 404 (*Not Found*).

O último *endpoint* testado foi o *BlockCard* que bloqueia o cartão via PUT, passando como parâmetro o número do cartão e os dados do usuário, todos válidos. Assim como qualquer requisição, foi utilizado o *token* antes de realizar a chamada de bloqueio do cartão. O resultado retornado pode ser observado na Figura 12.

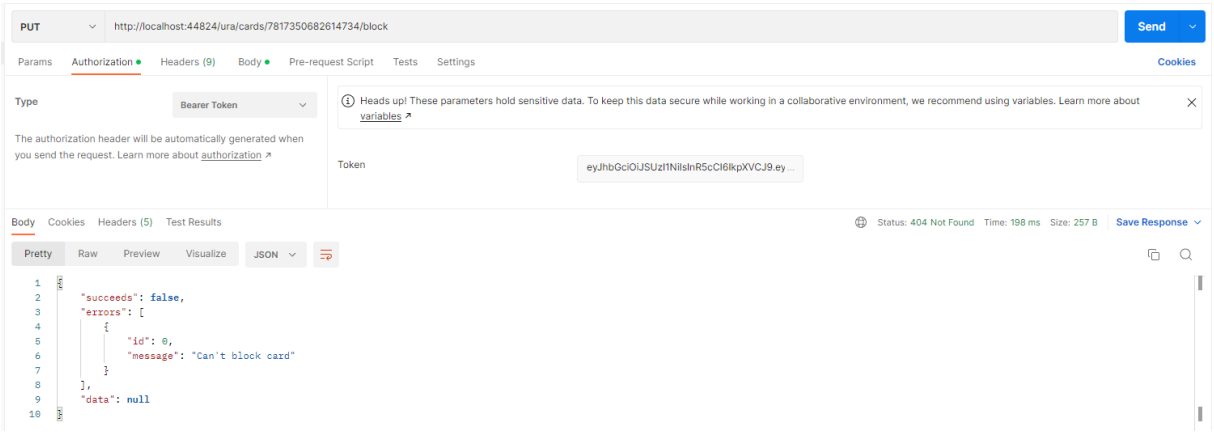
FIGURA 12 – TESTE FUNCIONAL 5



Fonte: próprio autor.

Nesse caso o bloqueio do cartão foi bem-sucedido e o status da operação retornado foi 204 (*No Content*), indicando que a solicitação foi bem-sucedida e o cliente não precisa sair da página atual. Realizando a chamada do *endpoint* novamente, com os mesmos parâmetros do cartão e do usuário, obteve-se o seguinte resultado, conforme ilustrado na Figura 13.

FIGURA 13: TESTE FUNCIONAL 6



Fonte: próprio autor.

Foi retornada uma mensagem em que o cartão não pôde ser bloqueado, com o status de operação 404 (*Not Found*), mesmo utilizando os parâmetros anteriores. Isso se deve a uma regra de negócio em que um cartão já bloqueado não pode ser bloqueado novamente. Nesse exemplo, além de ser realizado o teste funcional, foi executado também um teste de validação.

Conclui-se que os testes funcionais abrangem grande parte dos requisitos de negócios, pois validam as funcionalidades da API em diferentes cenários de uso. Uma grande vantagem é que podemos reutilizar os casos de testes em diferentes fases do desenvolvimento, poupando tempo e recurso.

Assim como no teste de validação, a utilização do Postman para esse tipo de teste se deve à simplicidade, rapidez e eficiência da ferramenta, permitindo a reutilização de solicitações para economizar tempo.

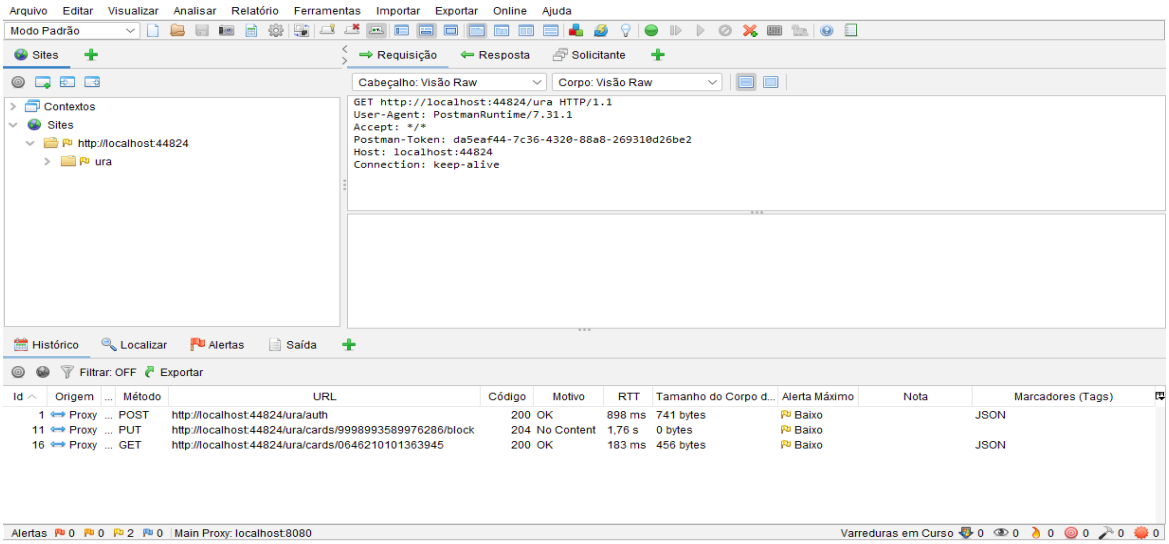
3.2.3 Teste de Segurança

Inicialmente, no desenvolvimento dos testes de segurança, foi analisado o código-fonte da API. Primeiramente, para ter acesso ao código-fonte e realizar a chamada da API localmente ou no ambiente de homologação, é necessário estar conectado na VPN (Rede Privada Virtual) com login e IP autorizados pela Up Brasil. Na análise do código foram buscadas vulnerabilidades como chamadas de funções não confiáveis e permissões que possam dar brecha, onde ambas não foram encontradas.

Com a aplicação sendo executada, foi analisado o comportamento dela através de chamadas dos *endpoints*, com determinadas entradas de parâmetros para observar como a aplicação se comporta, verificando também as suas saídas. Em relação a exposição de dados sensíveis na saída da aplicação, esse problema não foi identificado.

Ainda com a aplicação em execução, foi utilizado a ferramenta OWASP para realizar o teste de segurança na API, e os endpoints escolhidos foram o *Auth*, *GetCard* e *BlockCard*, conforme ilustrado na Figura 14.

FIGURA 14 – HISTÓRICO DE REQUISIÇÕES DO OWASP

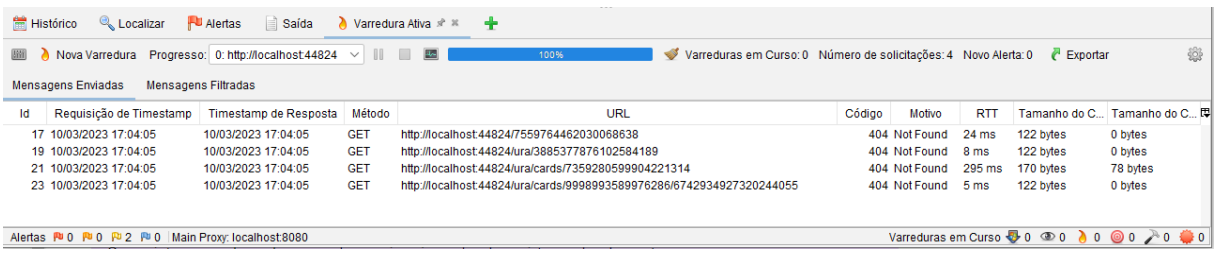


Fonte: próprio autor.

Conforme os *endpoints* são chamados, no caso, via Postman, as requisições ficam à mostra na aba de Histórico, detalhando o tipo de método, a URL do *endpoint* e o tempo de execução. Essa aba serviu para visualizar o registro de todas as solicitações e respostas HTTP que foram enviadas durante a varredura da aplicação *web* em busca de vulnerabilidades de segurança. Também foi útil para revisar e analisar as ações realizadas durante a varredura, bem como para identificar possíveis vulnerabilidades que foram encontradas.

A ação executada pela ferramenta foi a varredura ativa, também conhecida como Teste de Penetração (*Penetration Testing*), que é uma técnica de teste de segurança que visa simular um ataque real ao sistema e identificar possíveis vulnerabilidades, conforme mostrado na Figura 15. Vale ressaltar que o protocolo utilizado nesse teste foi o HTTP pelo motivo da API ter sido executada localmente, e o protocolo utilizado no ambiente de homologação/produção é o HTTPS, que é uma versão mais segura que utiliza um certificado SSL/TLS para criptografar as conexões.

FIGURA 15 – TESTE DE SEGURANÇA



Fonte: próprio autor.

Essa técnica envolve a realização de testes exploratórios, que são testes que não seguem nenhuma roteirização e nenhuma estratégia como BDD ou TDD, simplesmente a utilização do software, no caso o OWASP, com o objetivo de identificar pontos fracos na aplicação e avaliar o nível de resistência da mesma a possíveis ataques. E, como mostrado na figura, não foi encontrada nenhuma vulnerabilidade.

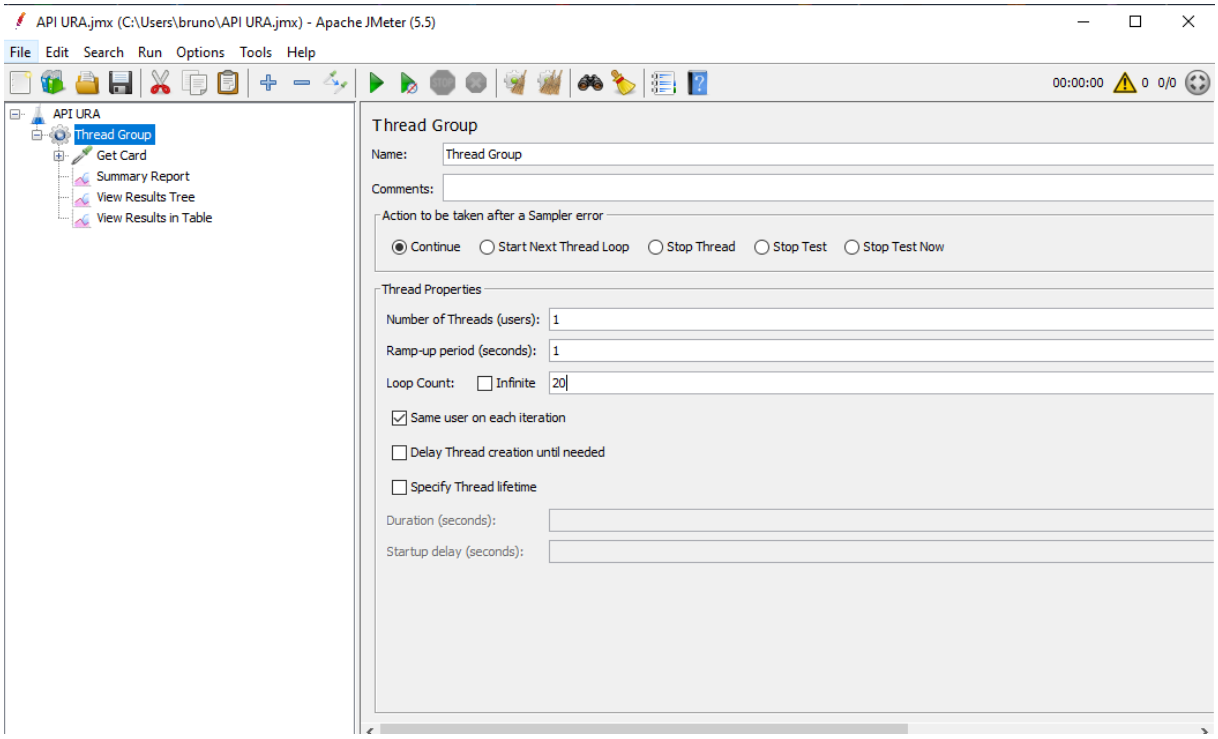
Esse tipo de teste foi importante para identificação de possíveis vulnerabilidades, como injeção de SQL e outros ataques comuns. Assim, foi possível garantir que os dados do usuário estão protegidos contra acesso não autorizado. A desvantagem consiste na complexidade de simular os cenários de ataque e na dependência de outras ferramentas mais especializadas.

A razão para a utilização da ferramenta OWASP para esse teste foi, primeiramente, pela credibilidade da organização, a qual é bem conhecida e respeitada na área de segurança de aplicativos *web*. Além de ser uma ferramenta com foco em segurança, possui suporte à automação e uma comunidade bastante ativa de usuários e desenvolvedores, permitindo encontrar suporte em grupos e fóruns.

3.2.4 Teste de Performance

Para o desenvolvimento do teste de performance foi utilizado a ferramenta JMeter, e o *endpoint* escolhido foi o *GetCard*. Com a aplicação sendo executada localmente, o teste foi configurado na sessão *Thread Group*, onde foi possível escolher o número de *threads* que foram utilizados na execução do teste; o tempo que levou para que os *threads* fossem criados (*ramp-up*); e a quantidade de vezes que foi executada as requisições, conforme ilustrado na Figura 16.

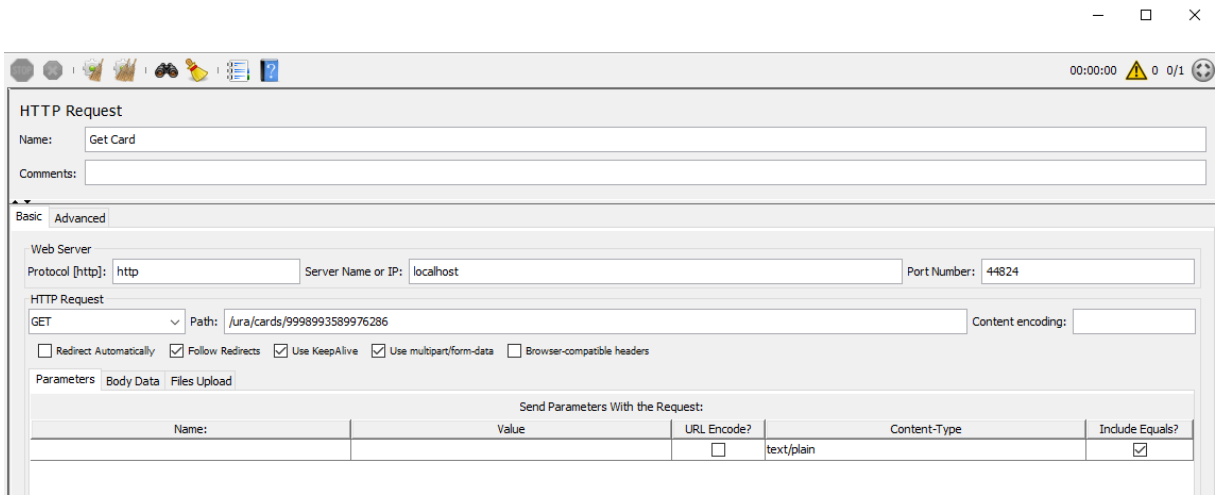
FIGURA 16 – CONFIGURAÇÃO DO TESTE DE PERFORMANCE



Fonte: próprio autor.

Após a configuração, a requisição foi configurada na sessão *HTTP Request*, mostrado na Figura 17.

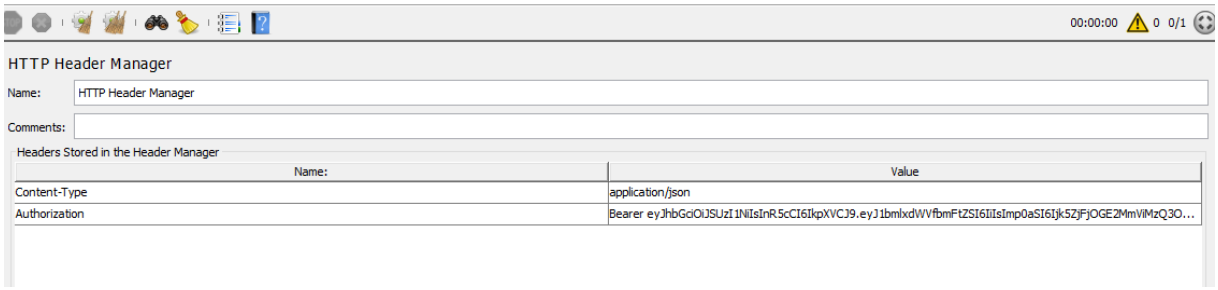
FIGURA 17 – CONFIGURAÇÃO DO HTTP REQUEST



Fonte: próprio autor.

Assim como nos outros testes, as chamadas dos *endpoints* necessitam no *token* de autenticação. Para adicionar o *token*, previamente gerado via Postman, foi utilizado uma sessão de cabeçalhos chamado *HTTP Header Manager*. O cabeçalho é usado para fornecer informações adicionais ao servidor, como o tipo de conteúdo e informações de autenticação, conforme mostrado na Figura 18.

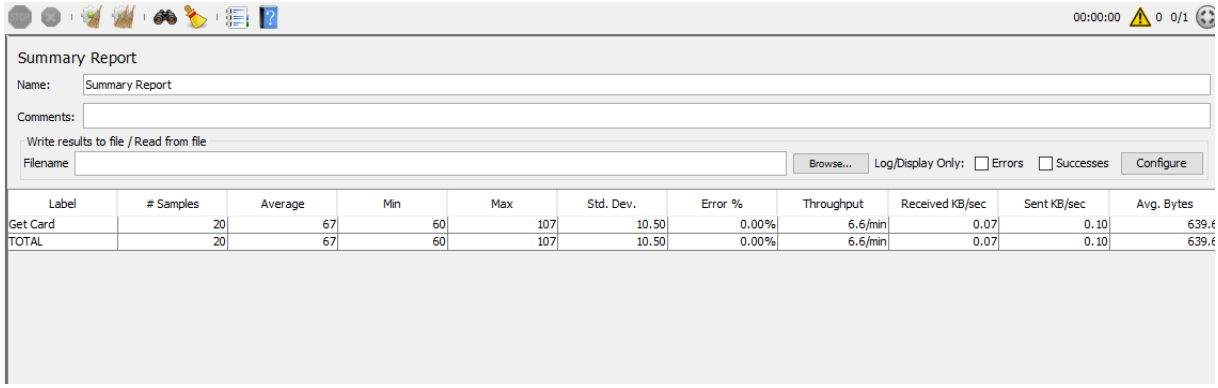
FIGURA 18 – CONFIGURAÇÃO DO HTTP HEADER MANAGER



Fonte: próprio autor.

Com o *HTTP Request* e o *HTTP Header Manager* prontos, foi iniciado o teste. Na sessão *Summary Report* foi possível visualizar um relatório em forma de tabela, facilitando a análise dos resultados. Conforme ilustrado na Figura 19, podemos ver o tempo mínimo, médio e máximo das 20 requisições feitas, além do *throughput* (transações por minuto), onde um valor alto significa que o sistema é capaz de lidar com um grande volume de tráfego, enquanto um valor baixo indica que o sistema pode estar sofrendo problema de desempenho.

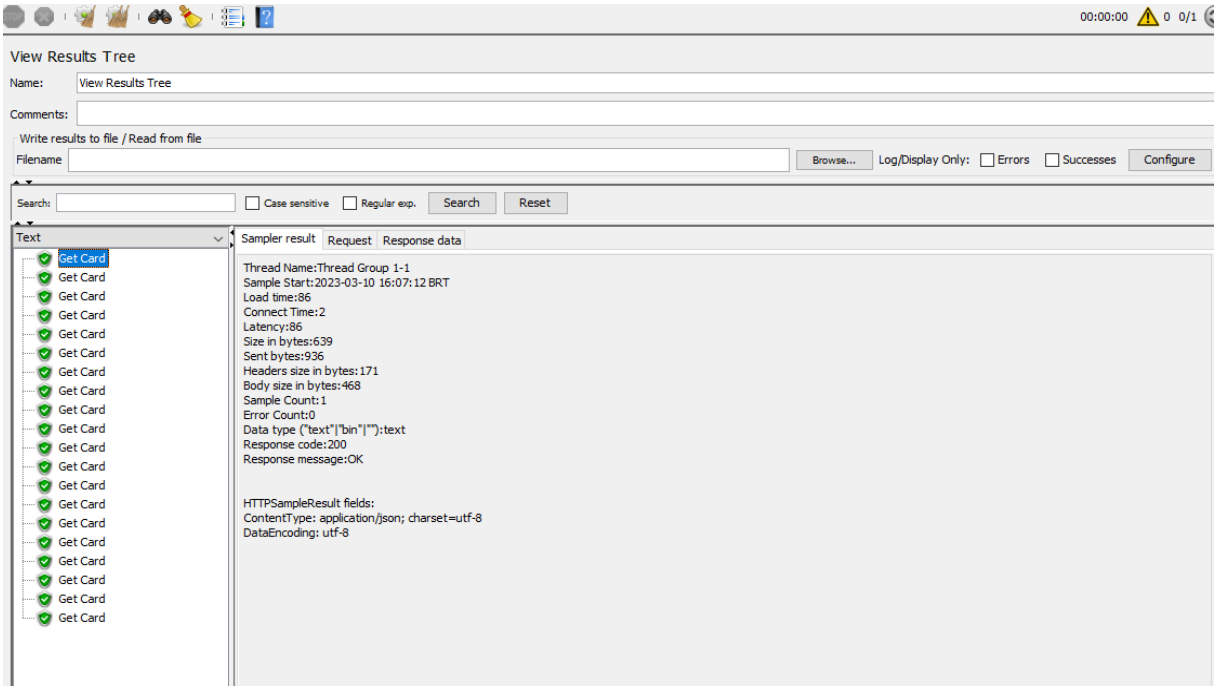
FIGURA 19 – SUMMARY REPORT



Fonte: próprio autor.

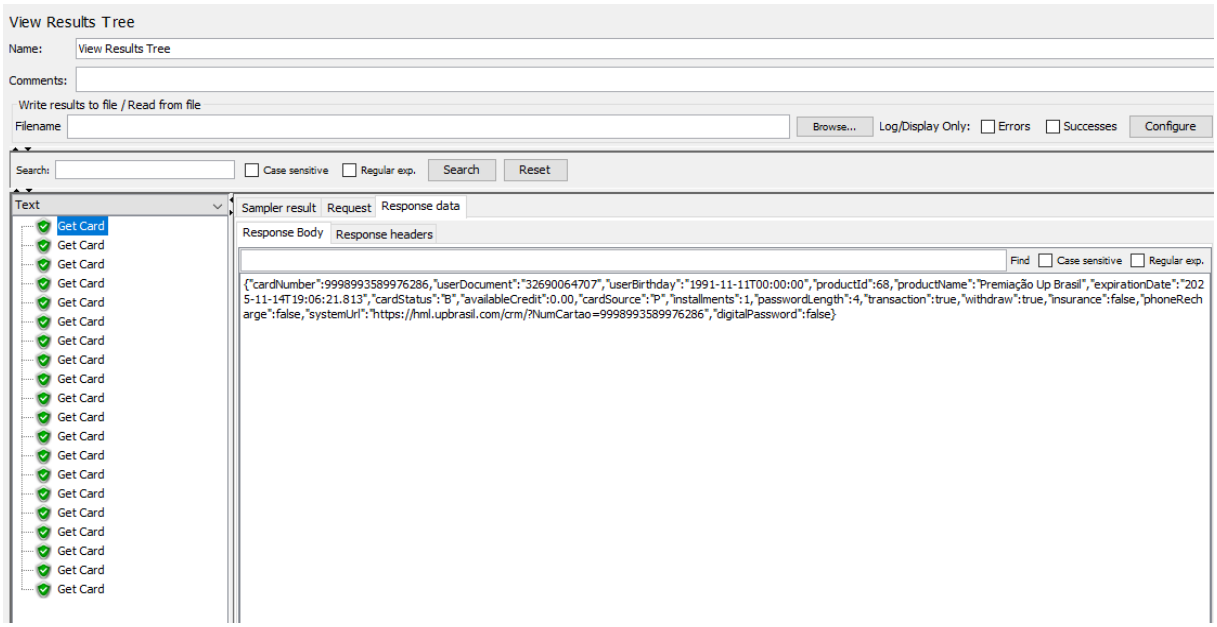
Na sessão *View Results in Tree* foi possível visualizar os resultados de cada amostra de solicitação enviada no teste, podendo analisar o desempenho de cada solicitação de forma individual. Conforme ilustrado na Figura 20 e 21, podemos ver a latência, o tempo de resposta e processamento, o status de operação e o corpo da resposta.

FIGURA 20 – VIEW RESULTS TREE – SAMPLER RESULT



Fonte: próprio autor.

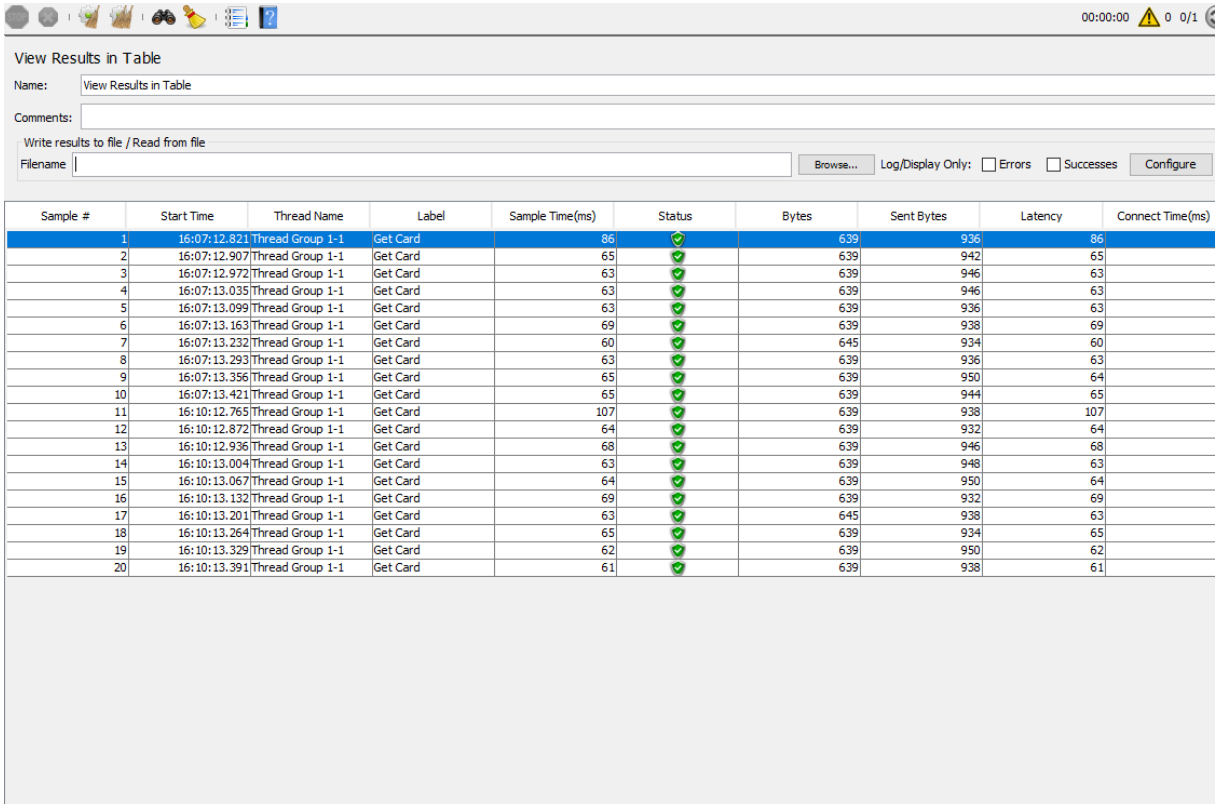
FIGURA 21 – VIEW RESULTS TREE – RESPONSE DATA



Fonte: próprio autor.

E, por último, a sessão *View Results in Table* exibe informações como o tempo de resposta de requisição, o código de status da resposta, o tamanho da resposta, entre outras, em forma de tabela, conforme ilustrado na Figura 22.

FIGURA 22 – VIEW RESULTS IN TABLE



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	16:07:12.821	Thread Group 1-1	Get Card	86		639	936	86	
2	16:07:12.907	Thread Group 1-1	Get Card	65		639	942	65	
3	16:07:12.972	Thread Group 1-1	Get Card	63		639	946	63	
4	16:07:13.035	Thread Group 1-1	Get Card	63		639	946	63	
5	16:07:13.099	Thread Group 1-1	Get Card	63		639	936	63	
6	16:07:13.163	Thread Group 1-1	Get Card	69		639	938	69	
7	16:07:13.232	Thread Group 1-1	Get Card	60		645	934	60	
8	16:07:13.293	Thread Group 1-1	Get Card	63		639	936	63	
9	16:07:13.356	Thread Group 1-1	Get Card	65		639	950	64	
10	16:07:13.421	Thread Group 1-1	Get Card	65		639	944	65	
11	16:10:12.765	Thread Group 1-1	Get Card	107		639	938	107	
12	16:10:12.872	Thread Group 1-1	Get Card	64		639	932	64	
13	16:10:12.936	Thread Group 1-1	Get Card	68		639	946	68	
14	16:10:13.004	Thread Group 1-1	Get Card	63		639	948	63	
15	16:10:13.067	Thread Group 1-1	Get Card	64		639	950	64	
16	16:10:13.132	Thread Group 1-1	Get Card	69		639	932	69	
17	16:10:13.201	Thread Group 1-1	Get Card	63		645	938	63	
18	16:10:13.264	Thread Group 1-1	Get Card	65		639	934	65	
19	16:10:13.329	Thread Group 1-1	Get Card	62		639	950	62	
20	16:10:13.391	Thread Group 1-1	Get Card	61		639	938	61	

Fonte: próprio autor.

Este teste ajudou a identificar possíveis gargalos na aplicação, ajudando a melhorar o desempenho e a eficiência da API. Com esse tipo de teste, também foi possível avaliar a capacidade da API de lidar com um número expressivo de solicitações simultâneas e, com essas informações, é possível garantir uma melhor experiência ao usuário com tempos de respostas mais rápidos. A principal desvantagem é em relação ao tempo de execução, que não se aplica a esse caso, porém o teste pode demandar um tempo significativo dependendo da complexidade da API.

Foi escolhida a ferramenta Apache JMeter para os testes de performance devido à sua flexibilidade em suportar vários protocolos HTTP, HTTPS, REST e muitos outros, tornando-o versátil para diferentes tipos de testes. A interface simples e seu relatório detalhado sobre o desempenho foram outros motivos para a escolha da ferramenta.

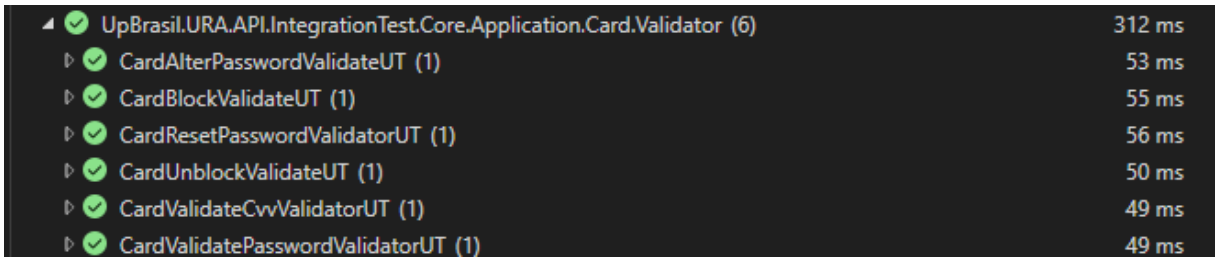
3.2.5 Teste de Integração

Para os testes de integração foram analisados os testes já existentes, que foram criados durante o desenvolvimento da API. Os testes de integração foram baseados na biblioteca xUnit para verificar se os diferentes componentes do sistema funcionam corretamente juntos, testando a integração entre as partes da API.

Pegando o cenário CardAlterPasswordValidate como modelo para demonstrar o teste integração, nesse cenário inicialmente foi realizado a autenticação do usuário pelo *endpoint* Auth, para gerar o *Bearer token.* Com o *token* gerado, foi chamado o *endpoint* GetCard para um cartão já existente e, com os dados do cartão retornados, foi chamado o *endpoint* AlterPassword para alterar a senha do cartão. Na sequência, foi chamado o ValidatePassword para verificar se o padrão de senha e, após validado, a senha foi alterada. Na última etapa do cenário, foi realizado uma consulta no BD (Banco de Dados) e foi verificado que a senha estava alterada conforme a descrição do cenário. Vimos que diferentes componentes que foram desenvolvidos de forma individual, funcionaram de forma conjunta, passando pelo teste de integração.

O xUnit, no caso, foi utilizado para escrever os testes que fizeram solicitações HTTP, manipularam dados e verificaram se as respostas da API estão corretas. Esses testes foram organizados em classes e métodos que representam diferentes cenários de teste, conforme ilustrado na Figura 23.

FIGURA 23 – TESTE DE INTEGRAÇÃO



Fonte: próprio autor.

Testes de validação de alteração de senha do cartão, validação de *reset* de senha, validação de bloqueio do cartão foram executados, obtendo sucesso em todos os cenários. Os detalhes do código fonte do teste foram omitidos devido a privacidade.

Ao fim dos testes, concluiu-se que o teste de integração permite checar a existência de problemas de integração entre os diferentes componentes, e que possui uma grande importância no ciclo do desenvolvimento da API. O empecilho encontrado durante o teste foi somente em relação à complexidade na criação dos cenários de testes.

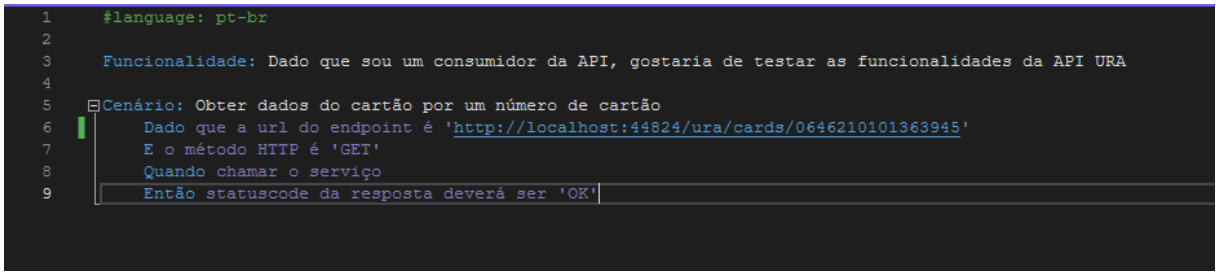
O uso do xUnit para esse tipo de teste foi pela sua facilidade de uso e por ser amplamente adotada e usada em várias linguagens de programação, no caso o C#, permitindo que os cenários de teste sejam escritos e executados de forma eficiente.

3.2.6 BDD

Nessa etapa do trabalho foi proposta a aplicação da técnica de *Behavior-Driven Development* (BDD) nos testes de API, com a utilização da ferramenta *SpecFlow* para automatizar os testes BDD com .NET.

Foi criado um projeto dentro da *solution* da API e foi selecionado o SpecFlow como *template* do projeto, especificando .NET 6 como *framework* a ser utilizado, juntamente com o framework de teste xUnit. Após a criação, foi adicionado um arquivo de *feature*, conforme ilustrado na Figura 24.

FIGURA 24 – FEATURE



Fonte: próprio autor.

O recurso *feature* do SpecFlow permite ao desenvolvedor descrever os cenários que devem ser testados e a condições de sucesso para cada cenário, que pode ser facilmente compreendida por pessoas não técnicas através de uma linguagem natural. No caso, foi criado um cenário em que dado uma *url* do *enpoint GetCard* com um número de cartão válido, o *statuscode* da resposta seja 200 (‘OK’), através da linguagem Gherkin.

Para que o cenário fosse executado, foi necessário criar uma implementação real de cada passo definido no *feature*. E essa implementação foi definida no *Step Definition*, que é

responsável por executar a lógica do teste, interagir com a API e realizar as assertivas necessárias para verificar se a funcionalidade está funcionando corretamente. Os passos são ilustrados conforme as Figuras 25 e 26.

FIGURA 25 – STEP DEFINITION 1

```
[Given(@"que a url do endpoint é '(.*)'")]
[Obsolete]
0 referências
public void DadoQueAUrlDoEndpointEHttpLocalhostUraCards(int url)
{
    ScenarioContext.Current["Endpoint"] = url;
}

[Given(@"e o método HTTP é '(.*)'")]
[Obsolete]
0 referências
public void EOMetodoHttpE(string p0)
{
    var metodo = Method.Post;

    switch (p0)
    {
        case "POST":
            metodo = Method.Post;
            break;
        case "GET":
            metodo = Method.Get;
            break;
        case "DELETE":
            metodo = Method.Delete;
            break;
        case "PUT":
            metodo = Method.Put;
            break;
        case "PATCH":
            metodo = Method.Patch;
            break;
        default:
            Assert.Fail("Método HTTP não esperado");
            break;
    }

    ScenarioContext.Current["HttpMethod"] = metodo;
}
```

Fonte: próprio autor.

FIGURA 26 – STEP DEFINITION 2

```
[When(@"chamar o servico")]
[Obsolete]
0 referências
public void QuandoChamarOServico()
{
    var endpoint = (string)ScenarioContext.Current["Endpoint"];

    ExecutarRequest(endpoint);
}

[Then(@"statuscode da resposta devera ser '(.*)'")]
[Obsolete]
0 referências
public void EntaoStatuscodeDaRespostaDeveraSer(string p0)
{
    var response = (RestResponse)ScenarioContext.Current["Response"];

    Assert.AreEqual(p0, response.StatusCode.ToString());
}

#region Private
[Obsolete]
1 referência
private RestResponse ExecutarRequest(string endpoint)
{
    var url = endpoint;
    var request = new RestRequest();

    request.Method = (Method)ScenarioContext.Current["HttpMethod"];

    var json = (string)ScenarioContext.Current["Data"];

    if(!String.IsNullOrEmpty(json))
    {
        request.AddParameter("application/json", json, ParameterType.RequestBody);
    }

    var restClient = new RestClient(url);
    var response = restClient.Execute(request);

    ScenarioContext.Current["Response"] = response;

    return response;
}

#endregion
```

Fonte: próprio autor.

Foram criados os métodos correspondentes para cada passo na *feature*, conforme detalhado abaixo:

- O primeiro método chamado DadoQueAUrlDoEndpointEHttpLocalhostUraCards é uma *function* que tem como parâmetro a *url*, que é uma *string*, e o armazena no *ScenarioContext*, que é uma classe estática que armazena dados durante a execução do cenário. Este método simula o passo “Dado que a url do enpoint é ‘http://localhost:44824/ura/cards/0646210101363945’”.
- O segundo método chamado EOMetodoHttpE é uma *function* que tem como parâmetro o método HTTP, que é uma *string*, e faz uma validação se esse parâmetro faz parte de

um método HTTP e o armazena no *ScenarioContext*. Este método simula o passo “E o método HTTP é ‘GET’”.

- O terceiro método chamado QuandoChamadoOServico acessa os dados armazenados nos métodos anteriores, no caso a *url* do *endpoint* e o método HTTP, e faz a requisição, retornando o seu *status code.* Este método simula o passo “Quando chamar o serviço”.
- O quarto método chamado EntaoStatuscodeDaRespostaDeveraSer tem como parâmetro o *status code,* que é uma *string,* e faz a comparação com o *status code* retornado no método anterior e, como ambos foram OK, o cenário está de acordo com o esperado.
- O atributo *Obsolete* foi implementado no código para gerar um aviso durante a compilação, indicando que aqueles cenários seria somente para este trabalho, pois o teste foi realizado do mesmo projeto de API URA.

4 LIÇÕES APRENDIDAS

Durante a realização deste trabalho de testes de API, várias lições valiosas foram aprendidas, destacando-se a importância de uma preparação antes de executar qualquer teste, que inclui a definição dos cenários de testes, a seleção do conjunto de dados e a escolha das ferramentas para garantir a eficácia do processo de teste.

A principal lição aprendida foi a importância dos testes de API no processo de criação de uma aplicação, pois com os testes é possível ter um retorno sobre a funcionalidade da nossa aplicação e conseguimos certificar que a comunicação entre a aplicação, sistemas e banco de dados estejam de acordo e, com isso, os testes criados servem como documentação para futuras validações.

Outro aprendizado importante é que deve ter uma preparação adequada para realizar os testes. Essa preparação envolve primeiramente entender a estrutura da API, os seus *endpoints*, seus parâmetros de entrada e a regra de negócio. Esta preparação foi fundamental para a criação de cenários de teste ser abrangente e detalhada. O estudo das ferramentas também foi de extrema importância para a realização dos testes.

Ao mesmo tempo que o ponto forte do trabalho foi a abranger diferentes tipos de testes, na qual conseguiu-se demonstrar uma compreensão das várias abordagens de testes disponíveis e a capacidade de aplicar esses testes, considero que também foi um ponto fraco do trabalho, pois não permitiu um foco adequado em cada tipo de teste.

Ao final dos testes, foi percebido que nem todos os tipos de testes são necessários para uma API, visto que nem sempre a necessidade de um tipo de negócio é igual a outra. Um exemplo é o teste de performance, caso a aplicação não possua uma grande demanda, os problemas de desempenho não são uma preocupação imediata. Deste modo, ficou o aprendizado de que não é estritamente necessário realizar todos os testes em uma API, depende dos requisitos e das características específicas da aplicação em questão.

Assim, um dos pontos de melhoria para um trabalho futuro seria aprofundar e explorar os tipos de testes de API, permitindo um detalhamento mais rico em informações, corrigindo o ponto fraco mencionado anteriormente. Outra possível melhoria seria um tópico dedicado à instalação das ferramentas utilizadas no trabalho e a utilização de uma API pública, possibilitando às pessoas realizarem os testes também.

5 CONCLUSÕES

Neste trabalho, foi apresentada uma análise dos tipos de testes de API, com foco na importância deles para garantir a qualidade de uma *API*. Além disso, foi abordado o conceito de *Behavior-Driven Development* (BDD) e como ele pode ser aplicado na realização de testes de *API*. Para a implementação do BDD, foram apresentadas algumas ferramentas, como SpecFlow e xUnit.

Os resultados indicam que os testes de API são fundamentais para garantir a qualidade de uma API, permitindo verificar se a aplicação atende aos requisitos especificados e se está funcionando corretamente. A utilização do BDD na realização desses testes traz diversos benefícios, como a maior clareza na comunicação entre desenvolvedores e clientes, o aumento da eficiência na detecção de *bugs* e a facilidade de manutenção dos testes.

A implementação do BDD foi realizada utilizando o SpecFlow e o xUnit, que permitiram escrever testes de integração em uma linguagem natural e com uma estrutura organizada. Comparado com os outros tipos de testes, o uso do BDD facilitou a compreensão e manutenção dos testes, enquanto o restante dos testes é mais técnico.

Em suma, a realização de testes de API é fundamental para garantir a qualidade de uma aplicação, e a utilização do BDD pode trazer diversos benefícios na realização desses testes. A escolha entre técnicas BDD e testes automatizados de API deve levar em consideração as necessidades específicas de cada projeto, podendo ser utilizadas em conjunto para obter melhores resultados.

Considerando o estudo realizado neste TCC sobre diferentes tipos de testes de API e a aplicação da abordagem BDD, um trabalho futuro pode se concentrar em detalhar mais os testes, desde a instalação das ferramentas até a parte prática dos testes, juntamente com o envolvimento de mais tipos de testes.

REFERÊNCIAS BIBLIOGRÁFICAS

API MIKE. **Wha tis API Validation**. Disponível em: <https://apimike.com/api-validation-a-guide>. Acesso em: 30 de ago. 2023.

BRAJESH, D. **API Management: An Architect's Guide to Developing and Managing APIs for Your Organization**. Apress. 1ª Ed. 2017

CRISPIN, L.; GREGORY, J. **Agile Testing: A Practical Guide for Testers and Agile Teams**. Addison-Wesley Professional. Edição Ilustrada. 2009.

CUCUMBER. **Especificação Gherkin**. Disponível em: <https://cucumber.io/docs/gherkin/reference/>. Acesso em: 05 de jul. 2023.

ELLIOT, E. Behavior Driven Development (BDD) and Functional Testing. Disponível em: <https://medium.com/javascript-scene/behavior-driven-development-bdd-and-functional-testing-62084ad7f1f2>. Acesso em: 05 de jul. 2023.

FIELDING, R. T**. Architectural Styles and the Design of Network-based Software Architectures**. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Acesso em: 12 dez. 2022.

Fowler, M. Integration **Test from Martin Fowler website**. Disponível em: <https://martinfowler.com/bliki/IntegrationTest.html>. Acesso em 13 fev. 2023.

MAILGUN. What Is a RESTful API, How It Works, Advantages and Examples. Disponível em: <https://www.mailgun.com/blog/it-and-engineering/restful-api/>. Acesso em: 05 de jul. 2023.

MASSÉ, MARK. **REST API Design Rulebook**. O'Reilly Media, 1ª Ed. 2011.

MICROSOFT. **Visual Studio 2022**. Disponível em: <https://visualstudio.microsoft.com/vs/>. Acesso em: 05 de jul. 2023.

MICROSOFT. **What's new in Visual Studio 2022**. Disponível em: <https://docs.microsoft.com/en-us/visualstudio/whats-new/visual-studio-2022/>. Acesso em: 05 de jul. 2023.

MOZZILA. **HTTP request methods – HTTP**. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. Acesso em: 30 de ago. 2023.

MYERS, Glenford J. et al. **The art of software testing**. Wiley, 3rd Revised ed. 2011.

NORDICAPIS. **10 Types of API Testing**. Disponível em: <https://nordicapis.com/10-types-of-api-testing/>. Acesso em: 29 de ago. 2023.

OWASP. (2022). **About the OWASP Foundation**. Disponível em: <https://owasp.org/about/>. Acesso em: 05 de jul. 2023.

POSTMAN. **Site oficial do Postman**. Disponível em: <https://www.postman.com/what-is-postman/>. Acesso em: 05 de jul. 2023.

Postman. **API testing**. Disponível em: <https://learning.postman.com/docs/writing-scripts/test-scripts/>. Acesso em: 11 fev. 2023.

SMART, John. **BDD in action: Behavior-driven development for the whole software lifecycle**. Manning Publications, 1ª Ed. 2014

SMARTBEAR. **What Is API Security Testing?** Disponível em: <https://smartbear.com/learn/api-testing/security/>. Acesso em: 05 de jul. 2023.

SPECFLOW. **Site oficial do SpecFlow**. Disponível em: <https://specflow.org/>. Acesso em: 05 de jul. 2023.

TERRA, JOHN.**What is Integration Testing?** Disponível em: <https://www.simplilearn.com/what-is-integration-testing-examples-challenges-approaches-article>. Acesso em: 05 de jul. 2023.

THALAYASINAAM, C. **API Testing 101**. Disponível em: <https://blog.testproject.io/2021/06/16/api-testing-101/>. Acesso em: 06 de jul. 2023.