SISTEMA BAEP: ESTUDO DE TESTES E MÉTRICAS PARA A QUALIDADE DE SOFTWARE

MUTA, Davi Massaru Teixeira. OLIVEIRA, Lucas Otaviani Bergamo de.
VOLPE, Valéria Maria. SILVA, Djalma Domingos da.

e-mail:
davimassaru@hotmail.com; lucas.otaviani@hotmail.com; vmvolpe@fatecriopreto.edu.br;
djalma@fatecriopreto.edu.br

Resumo: O presente trabalho descreve o processo de desenvolvimento back-end do sistema BAEP, aplicando técnicas para garantir uma maior qualidade de software, como o uso das métricas de halstead, testes de software e testes de mutação, entregando um sistema que seja capaz de informatizar o processo que a 9ª BAEP (Batalhão de Ações Especiais da Polícia) possui no preenchimento manual do relatório de serviço operacional, unificando os dados em uma base de dados que oferecendo maior controle para os policiais.

Palavras-chave: Testes de mutação. Qualidade de software. Métricas de Halstead. Sistema BAEP.

Abstract: *The present work, the back-end development process of the BAEP system, applying techniques to ensure greater software quality, such as the use of halstead metrics, software testing and mutation testing, delivering a system capable of computerizing the process that the 9th BAEP (Battalion of Special Actions of the Police) does not have manual filling of the operational service report, unifying the data in a database that offers greater control to the police.*

Keywords: *Mutation testing. Software quality. Halstead Metrics. BAEP system.*

1 Introdução

Com a necessidade da emissão do "relatório de serviço operacional" do 9º BAEP (Batalhão de Ações Especiais da Polícia), se fez necessário o desenvolvimento de um sistema de apoio, com a finalidade de automatizar o preenchimento do relatório e utilizar os dados coletados para colaborar em tomadas de decisões estratégicas da polícia de São José do Rio Preto.

Para desenvolvimento do projeto, foi realizado o levantamento de requisitos funcionais e não funcionais do sistema, a definição do diagrama entidade-relacionamento do banco de dados a qual permitiu a criação das tabelas e o relacionamento entre os dados, assim como a definição dos atributos de cada entidade, além do diagrama de classes, que colaborou diretamente para a definição das classes, métodos e definição da arquitetura de *software*.

A partir do escopo do projeto definido, o projeto foi pensado para ser um projeto escalável, manutenível e seguro. Com tal premissa, estudos foram feitos para definir métricas

de aceitação de código, lidando com a análise estática, testes de *software*, garantindo cobertura de código, estabilidade do produto, irritabilidade – um dos pilares da segurança da informação – por meio de permissões de usuários, autenticação, *logs* e registro de histórico.

2 Justificativa

O desenvolvimento do projeto se deu pela necessidade do 9º BAEP de construir um *software* que os auxiliassem no desenvolvimento dos relatórios diários. Em entrevista com o capitão Táparo, responsável pelo acompanhamento do projeto, "a análise dos dados gerados pelos relatórios de serviço operacional, é feito de maneira primitiva". Atualmente os dados são analisados a partir de um documento físico preenchido a mão, emitidos a partir da ronda das viaturas, como pode ser visto na Figura 1. Estes, por sua vez, têm seus dados transferidos para um editor de planilha eletrônica. Todo o processo manual gera retrabalho pela equipe que o faz, causando possíveis inconsistências, além de possuir limitações de análise.

Figura 1 Relatório de serviço operacional

POLÍCIA MILITAR DO ESTADO DE SÃO PAULO
COMANDO DE POLICIAMENTO DO INTERIOR – 5
9º BATALHÃO DE AÇÕES ESPECIAIS DE POLÍCIA

RELATÓRIO DE SERVIÇO OPERACIONAL

Data: ___/___/___ Horário das ___ às ___ Hs ___ Cia. ___º Pelotão

VIATURA | COMPONENTES DA EQUIPE (Cmt, RE, Nome de Guerra; Mot; Seg; Seg; Seg) | RONDAS (HORA, RONDANTE)

Prefixo
Km Término
Km Início
Km Percorrido

PRODUTIVIDADE DA EQUIPE

Pessoas Vistoriadas		Munições Apreendidas		Tráfico de Drogas		Flagrantes	
Veículos Fiscalizados		Armas Fogo Total		Porte / Uso de Drogas		Presos em Flagrante	
Motos Fiscalizadas		Revólver		Maconha Kg		Ato Infracional	
Ônibus Vistoriados		Pistola		Cocaína Kg		Menores Apreendidos	
AIT Confeccionados		Carabina		Crack Kg		Condenados Capturad	
Veículos Apreendidos		Outras:		Outros Kg		Menores Capturados	
Doc/CLA Apreendidos				Dinheiro em Espécie R$		Escoltas Realizadas	
Estab. Comerciais				Obs.:		BO.e Confeccionados	
Auxílio ao Público						Veículos Recuperados	

OCORRÊNCIAS ATENDIDAS

Nº BOe	COD	NATUREZA	BTL e Cia	LOCAL	HORA
1					
2					
3					
4					

VEÍCULOS E/OU PESSOAS ABORDADAS

Placa / Ocorrência	NOME	DOC	LOCAL	HORA
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

APOIOS PRESTADOS

VIATURA	LOCAL	HORA	OBSERVAÇÕES
1			
2			
3			
4			

NOVIDADES COM A VIATURA

INSTRUÇÃO MINISTRADA

HISTÓRICO

Cmt Equipe:

Cmt Pelotão:

Fonte: Disponibilizado pelo 9º BAEP.

Hoje em dia, desenvolver um *software* escalável, manutenível e seguro é um desafio comum para qualquer programador, e por conta destes tópicos, várias metodologias e técnicas foram desenvolvidas para conduzir a produção de *software*. Tendo isto em mente, produzir um sistema robusto e flexível, utilizando boas práticas de programação como SOLID e Kiss, aplicando conceitos de engenharia e arquitetura de *software* não é mais um tabu, e sim uma necessidade.

Durante a elaboração do sistema, como forma de assegurar a manutenção do código, foi tomado como base a PSR-12 (Padrão de estilo de código em PHP) e a análise estática para rastrear eventuais problemas de código.

Como forma de garantir a integridade de cada ação do programa, foram implementados testes de *software* além de testes de mutação para averiguar a confiabilidade dos testes desenvolvidos.

3 Objetivos

O objetivo deste projeto foi projetar uma aplicação *back-end*, programando uma REST API a partir dos principais conceitos de engenharia de *software* e segurança da informação, desenvolvendo os requisitos do sistema, contemplando várias regras de negócios mapeadas, garantindo a qualidade do *software* por meio de métricas a fim de comprovar sua legibilidade, complexidade, eficácia contra *bugs* e a implementação de estratégias que garantam a irritabilidade.

4 Fundamentação Teórica

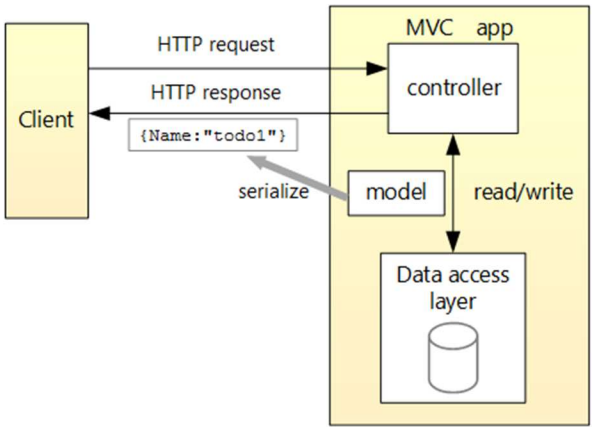
Neste tópico será abordado sobre os assuntos que envolveram sobre a fundamentação do projeto, dando ênfase na arquitetura em conjunto com as definições das tecnologias utilizadas.

Este projeto foi desenvolvido para utilizar a interface de aplicação denominada REST (*Representational State Transfer* – transferência de estado representacional em tradução livre), de acordo com a Red Hat (2020) “REST não é um protocolo ou padrão, mas sim um conjunto de restrições de arquitetura. Os desenvolvedores de API podem implementar a arquitetura REST de maneiras variadas.” a partir de tal premissa, o projeto foi projetado para desenvolver uma API – “Uma API é um conjunto de definições e protocolos usados no desenvolvimento e

na integração de aplicações." (Red Hat, 2020) – auxiliando na comunicação entre *back-end* e *front-end*.

A arquitetura utilizada no projeto, foi a *MVC* (*Model, View, Controller* – Modelo, Visualização, Controladora em tradução livre). Segundo Guedes (2020) "O MVC é um padrão de arquitetura de software. O MVC sugere uma maneira para você pensar na divisão de responsabilidades, principalmente dentro de um software web.", este modelo foi utilizado visto que o *framework* Laravel adota este modelo por padrão em seus projetos. A Figura 2 representa um esquema gráfico de como a arquitetura se comporta com seus devidos fluxos.

Figura 2 Arquitetura REST API com MVC



Fonte: Microsoft, 2021.

Em suma, um cliente realiza uma requisição para uma controladora, que por sua vez irá realizar uma ação através de um modelo, seja ler ou escrever uma informação no banco de dados, após a ação ser concluída, a controladora realiza uma a formatação da resposta da API para o cliente – representado pela definição "*serialize*" do modelo. Vale ressaltar que a camada de visualização foi a única não utilizada pois o *front-end* consumirá o retorno da API em formato JSON (*JavaScript Object Notation* – Notação de objeto JavaScript).

4.1 Metodologias e Tecnologias Utilizadas

Para garantir a estabilidade do *software*, e que determinadas respostas da API devem ser enviadas corretamente, foram aplicados testes de caixa branca e de caixa preta como forma de assegurar todos os caminhos possíveis, evitando a introdução de *bugs* em cenários comuns de uso, certificando que qualquer alteração possa impactar um cenário de teste já previsto.

4.1.1 Testes de Software

Testes de software podem ser definidos como técnicas para verificar se o comportamento de uma determinada funcionalidade, validando o seu comportamento conforme determinadas entradas de valores.

De acordo com NETO, 2015 “De uma forma simples, testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado.”

Testes de caixa-preta desconsideram o comportamento interno do programa, onde os dados de entrada são enviados à função e o resultado de ser previamente conhecido.

Testes de caixa-branca desconsideram o comportamento interno do programa, onde os testes têm acesso ao código fonte do módulo a ser testado, podendo fazer teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos. (NETO, 2015).

4.1.2 Cobertura de código

Testes de cobertura são um tipo de teste de caixa-branca, onde é verificado se os testes executados estão passando por determinados blocos a nível de código-fonte.

“A análise de cobertura de código é um tipo de técnica usada em teste de caixa-branca, cujo objetivo é verificar como o conjunto de casos de testes exercita partes do código. Portanto, é utilizada para averiguar a qualidade do conjunto de casos de testes e não a qualidade do produto real, principalmente na fase de teste de unidade, no qual requer a cobertura do código e dos caminhos possíveis dentro de cada unidade do programa.” (SOARES e VASCONCELOS, 2006).

4.1.3 Testes de Mutação

O teste de mutação envolve a modificação de um programa em pequenas partes, cada versão mutada é chamada de ‘mutante’. Para avaliar a qualidade de um determinado conjunto de testes, esses mutantes são executados em relação ao conjunto de testes de entrada, caso não ocorra a quebra de cenários de testes, significa que o mutante não foi percebido e, portanto, o código original deve validado.

4.1.2 Análise Estática

Analisar o código estaticamente colabora com uma visualização abrangente de tipos de variáveis e seus possíveis comportamentos, visibilidade dos métodos e garantia de padronização de códigos impostos pela comunidade. A análise estática é de grande importância para linguagens interpretadas como o PHP, segundo Mattos (2018), diferente de linguagens compiladas como Java e C# que contam com um compilador, atuando como uma "primeira linha de defesa", já que precisam saber sobre o tipo de cada variável, o tipo de retorno de cada método, antes da execução do programa. O compilador precisa ter certeza de que o programa está "correto".

4.1.2.1 PSR-12

Um padrão comum de análise estática é relacionada a estilização do código, a PSR-12 (*PHP Standards Recommendations*) é um padrão de estilo de código imposta pela comunidade PHP, que visa padronizar a formatação do projeto para colaborar com o desenvolvimento, validando indentação, espaçamento entre linhas, quebra de linhas, localização de chaves, dentre outras validações são aplicadas.

De acordo com php-fig, "PSR-12 foi aceito em 2012 e, desde então, várias alterações foram feitas no PHP, o que tem implicações nas diretrizes de estilo de codificação. Embora o PSR-2 seja muito abrangente em relação às funcionalidades do PHP que existiam no momento da escrita, a nova funcionalidade está muito aberta para interpretação"

4.1.3 Métricas de Halstead

Métricas de Halstead, são métricas de código desenvolvidas por Maurice Halstead (1977), para determinar o esforço mental necessário para desenvolver ou manter um programa, em sua essência, quanto menor os valores, menor será a dificuldade de realizar alterações no *software*,

"Halstead não fornece valores limites para suas métricas, sugerindo que 260 seria um valor adequado para o tamanho de um módulo (soma total de operandos e operadores)." (BAHIA,1992, p. 80)

As métricas de Halstead são geradas a partir da quantidade de *operators* e *operands*, que são definidas por:

- *Operator:* são todos os identificadores que não são palavras reservadas da tecnologia, como declaração de variáveis, constantes, valores numéricos ou *String*.
- *Operands*: palavras reservadas que qualificam tipo, operadores, além estruturas de controle como *if*, *else*, *for*, *while* e etc.

Métricas de Halstead são calculadas por:

- (n1) Número de *operators* distintos;
- (n2) Número de *operands* distintos;
- (N1) Número total das instâncias de *operators*;
- (N2) Número total das instâncias de *operands*.

Sendo elas:

Vocabulary (n) (Fórmula 1)

$$n = n1 + n2 \quad (1)$$

Está relacionado a complexidade do vocabulário definido dentro do programa sendo a soma distintas do *operators* e *operands*;

Size (N) (Fórmula 2)

$$N = N1 + N2 \quad (2)$$

É o tamanho do software em si, dado pela soma do número total das de instâncias de *operators* e *operands*;

Volume (V) (Fórmula 3)

$$V = N * log2\ (n) \quad (3)$$

É o conteúdo de informação do programa, medido em bits, calculado através do tamanho (N) do programa multiplicado pelo logaritmo de base 2 do vocabulário, baseando-se no número de operações realizadas e operandos tratados no algoritmo, por convenção caso o volume (V) seja maior que 1000, é um forte indício que a função provavelmente tem mais responsabilidades do que deveria, dificultando uma correção.

Difficulty (D) (Fórmula 4)

$$D = (n1 / 2) * (N2 / n2) \quad (4)$$

Proporcional ao número de operadores distintos e a razão entre o total das de instâncias de operandos (N2) e operandos distintos (n2), quando um programa utiliza os mesmos operandos com incidência, ele se torna mais suscetível a erros, pois os operandos possivelmente têm uma mutabilidade muito alta dentro da aplicação, dificultando a leitura e entendimento do código e diminuindo sua previsibilidade durante a execução do algoritmo.

Effort to implement (E) (Fórmula 5)

$$E = V * D \tag{5}$$

O esforço de implementação, é o esforço de compreensão do algoritmo, sendo a multiplicação entre o volume (V) e dificuldade (D).

Time to implement (T) (Fórmula 6)

$$T = E / K \tag{6}$$

O tempo de implementação ou de entendimento do programa é dado pela razão entre o esforço de entendimento pelo coeficiente K, que tem um valor padrão arbitrário de 18, mas que pode ser alterado de acordo com a experiencia dos desenvolvedores envolvidos, tendo como resultado um valor em segundos.

Errors (B) (Fórmula 7)

$$B = V / M \tag{7}$$

Estimativa de erros na implementação, sendo Volume dividido por M onde M se é a média de números de decisões entre erros, 3000 de acordo com Maurice Halstead.

No projeto foram definidos como valores de referência das métricas de acordo com o Quadro 1, extraído do trabalho de Emanuel Santos 2008 em seu trabalho de mestrado “Uma proposta de métricas para avaliar modelos i*”.

Quadro 1 Intervalo de aceitação das métricas

Métrica	Intervalo de aceitação		
	Aceita	Marginal	Rejeitada
Complexidade ciclomática (CC)	10 <= CC	10 < CC <=15	CC > 15
Vocabulary (n)	70 <= n	70 < n <= 120	n > 120
Size (N)	120 <= N	120 < N <= 150	N > 150
Volume (V)	800 <= V	800 < V <=1000	V > 1000
Difficulty (D)	4 <= D	4 < D <= 7	D > 7
Effort to implement (E)	E <= 1000	1000 < B <= 1500	E > 1500
Errors (B)	B <= 5	5 < B <= 10	B > 10

Fonte: Santo, 2008.

4.1.4 Complexidade Ciclomática

A complexidade ciclomática, é uma métrica referente ao quão complexo um bloco de código é, tem seu cálculo baseando-se na teoria dos grafos, quanto maior o mais complexo o algoritmo é.

De acordo com McCabe, 1976, a complexidade ciclomática de um *software* é dada pela diferença entre a quantidade de arestas (E) pela quantidade de nós (N) do grafo, somada duas vezes a quantidade de módulos conectados (P) (Fórmula 1).

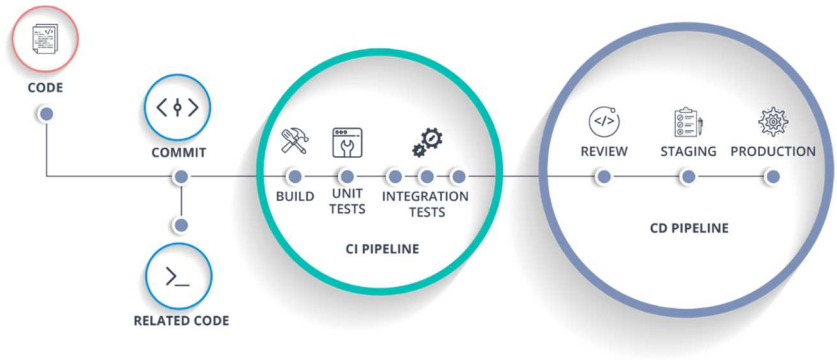
$$CC = E - N + 2P \quad (8)$$

4.1.5 Integração contínua

A ideia de manter o *software* testado por meio de testes unitários e padronizado pela análise estática, é garantir a consistência da aplicação. Sendo assim, foi previsto que qualquer

alteração possa impactar eventuais mudanças que acarretem possíveis problemas no comportamento padrão do *software*.

Figura 3 Processo de integração e entrega contínua



Fonte: Balajee, 2019.

Através de técnicas de integração e entrega contínua, conhecido popularmente por CI/CD, é possível evitar que anormalidades sejam disponibilizadas. Essa prática gera a automatização de serviços que podem lidar com validações de testes, análise estática e o deploy da aplicação, como demonstra a Figura 3.

4.1.6 Segurança da Informação

Testar, manter o código padronizado e ter um *pipeline*, garantem a estabilidade do *software* quando bem utilizados, no entanto, contar apenas com estes recursos não garantem a segurança dos dados, para isso, foram adicionadas algumas camadas de validações, definidas por *middlewares* e *policies*.

4.1.6.1 Middlewares

O *middleware* é a primeira camada de segurança que adota o padrão de projeto denominado "*Chain of Responsability*", cada um dos *middlewares* possuem uma responsabilidade, validar se há um usuário autenticado, se um *token* não está expirado etc. Sendo assim, cada uma das classes realiza uma validação, em caso de sucesso, o *middleware* encaminha a mensagem para o próximo, e assim sucessivamente, até que em um caso de falha, retorna uma mensagem de erro antes mesmo de executar qualquer processo na *controller*. (LARAVEL,2021)

4.1.6.2 Policies

As *policies* validam as permissões de um usuário autenticado ou de um convidado, verificando se o usuário possui autorização para realizar uma determinada ação, como por exemplo, ter acesso de leitura ou escrita a determinados *models*.

4.1.6.3 Logs

Logs, segundo a Rock Content (2019) "[...] é um arquivo que, geralmente, tem a extensão .log. [...] Ele contém informações sobre o que ocorre durante a operação de variados tipos de sistemas, entre eles: sistema operacional, banco de dados, site na internet e muitos outros.".

4.2 Tecnologias

Os próximos tópicos abordam as tecnologias que foram utilizadas no desenvolvimento do projeto.

4.2.1 PHP

O *PHP* (acrônimo para *Hypertext Preprocessor*), é uma linguagem de script interpretada open *source*, que pode ser utilizada de forma geral, ou seja, em *software*, páginas Web e processamento de dados, automatizando a execução de tarefas em um ambiente apropriado. A linguagem foi desenvolvida por Rasmus Lerdorf, em 1994 e escrita em C, a linguagem possui uma presença notável no desenvolvimento Web, com o intuito de construir sites dinâmicos no *server-side*, o que garante a confiabilidade dos dados que são enviados para o *front-end* da aplicação.

Cerca de 80% dos sites que conhecemos possuem a influência do PHP em seu escopo, a linguagem tomou conta da Web por conta da sua arquitetura de grande performance para sites.

4.2.2 Laravel

De acordo com o site oficial do Laravel (2021) "*Laravel is a web application framework with expressive, elegant syntax*." – Laravel, é um *framework* de aplicação web com sintaxe expressiva e elegante (em tradução livre) – ele fornece estrutura para criar uma aplicação web de forma fácil com ênfase nos detalhes, simplificando estruturas que poderiam ser complexas

de serem desenvolvidas, ele combina os melhores pacotes do ecossistema do *PHP*, para disponibilizar o *framework* mais robusto e mais amigável possível.

Segundo a w3scools (c2021) "*Laravel was developed and created by Taylor Otwell as an attempt to give an excellent substitute for the older PHP framework named CodeIgniter.*" – Laravel foi desenvolvido e criado por Taylor Otwell como uma tentativa de oferecer um excelente substituto para o antigo framework PHP chamado CodeIgniter (em tradução livre).

Através de um benchmark que compara os *frameworks back-end* desde 2011 até setembro de 2021, o Laravel se tornou o *framework* mais popular dentre as demais tecnologias (*Statistics and data,* 2021), demonstrando seu impacto na comunidade de desenvolvedores.

4.2.3 GraphQL

O *GraphQL* (*Graph Query Language*) de acordo com o site oficial (c2021) foi projetado e desenvolvido pelo Facebook em 2012 e tornado para uso público apenas em 2015. A tecnologia permite que os clientes definam a estrutura dos dados necessários para que sejam consultados do servidor, evitando um retorno excessivo de dados que não serão utilizados.

Sua arquitetura permite que o desenvolvedor tenha dados previsíveis ao buscar a resposta da *API*, isso porque o *GraphQL* trabalha com *schemas*, um conceito que representa um objeto e os *types* que defininem os dados de cada entidade e seus possíveis tipagens, permitindo que um *fetch* (uma consulta para validação) seja feita.

Em suma, o servidor permite o acesso para apenas uma única rota: a "/graphql", pois nela será executada uma query requisitada pelo *front-end* através de *Queries* ou *Mutations*.

A fim de utilizá-lo em conjunto com a arquitetura *back-end* fornecida pelo Laravel, o *framework Lighthouse* é o responsável por fornecer a rota, disponibilizar os *schemas*, os *types*, as *queries* e as *mutations*, interpretar e executar as consultas feitas na linguagem e aplicar os padrões de segurança.

4.2.4 PHP Unit

O *PHPUnit* é um *framework* desenvolvido por Sebastian Bergmann na linguagem PHP, e sua construção foi baseada no *xUnit* (nome genérico para se referir a um *framework* de testes unitários para qualquer linguagem de programação). O *xUnit* padroniza as asserções dos testes para que sejam semelhantes em qualquer linguagem, elas colaboram com a garantia de um

determinado output a partir de uma expectativa, garantindo então o comportamento padrão de uma *feature* ou uma função pelo input. (PHPUNIT, 2021).

Para escrever os testes do projeto, o *PHPUnit* foi o *framework* escolhido por possuir maior popularidade dentre os *frameworks* de testes do *PHP*. Para que todos os cenários de testes fossem cobertos, foram estabelecidas duas formas de testes:

- Testes Unitários;
- Testes de *Feature*;

Os testes unitários são os responsáveis por testar a menor unidade de código do sistema, ou seja, cada função deve ser testada com entradas válidas e inválidas para garantir que uma ação esperada deve ser executada.

Os testes de *feature* testam uma única funcionalidade como um todo, e não se preocupa com a menor unidade do código, simulando a experiência do usuário ao tentar realizar uma determinada ação.

4.2.5 *Infection*

O Infection é um pacote desenvolvido para aplicações em *PHP*, para executar testes de mutações. De acordo com a documentação oficial do *Infection* (c2021), esse tipo de teste é uma técnica baseada em falha que fornece um critério chamado *Mutation Score Indicator* (*MSI*), este parâmetro é utilizado para medir a eficácia de um conjunto de testes em termos de sua capacidade de detectar falhas.

Se o programa mutado produz testes com falha, isso é chamado de mutante morto, caso os testes estiverem verdes com código mutado, então temos um mutante com *escape*.

4.2.6 Larastan

O pacote “nonomaduro/larastan” realiza uma validação de análise estática, ou seja, que não executa o código para gerar resultados que informam possíveis problemas que podem ocorrer com o código ao ser executado, em especial, com a tipagem das variáveis, validando condicionais não coesas, chamadas de métodos indevidas, tipagens indefinidas ou incoerentes etc.

4.2.7 PHP Mess Detector

O *PHP Mess Detector* é um pacote que analisa estaticamente o código a fim de reportar problemas relacionados a qualidade de código, garantindo que boas práticas de programação sejam aplicadas no código, contendo os seguintes conjuntos de regras:

- Regras de código limpo
- Regras de tamanho de código;
- Regras controversas;
- Regras de design;
- Regras de nomenclatura;
- Regras de códigos não utilizados.

4.2.11 MySql

O MySql é um sistema de gerenciamento de banco de dados (SGBD) relacional, disponibilizado de forma gratuita e livre. Criado em 1995 por David Axmark e Michael Widenius, ambos engenheiros, desenvolveram a ferramenta a partir de uma necessidade própria, por sua interface simples e pela capacidade de rodar em vários sistemas operacionais, não demorou muito para que a ferramenta se popularizasse, e fosse recebendo atualizações da comunidade.

4.2.12 Docker

Docker é uma tecnologia desenvolvida a fim de executar aplicações virtualizadas a nível de sistema operacional, desenvolvida por Solomon Hykes em 2013 em linguagem Go e mantida pela Docker Inc.

Uma aplicação pode conter inúmeras ‘imagens’ que podem ser descritas como um *template* padrão para uma tecnologia. Com a junção das imagens escolhidas, é possível criar um ‘container’, ambiente que irá conter todas as imagens definidas a fim de unificar versões de tecnologias e definir um único ambiente de desenvolvimento.

A conteinerização de cada uma das tecnologias colabora para o crescimento de um ambiente de desenvolvimento único, o que significa que cada ambiente de trabalho permanece idêntico aos demais por atuar em cima de uma máquina virtual e independente do sistema operacional.

4.2.13 Nginx

O Nginx é um servidor web de código aberto e realiza a comunicação HTTP entre cliente e servidor, lidando com proxy reverso, load balancer e proxy de email para SMTP, POP3 e IMAP. Seu lançamento foi em outubro de 2004 e foi desenvolvido por Igor Syssoev, através de um desafio para gerenciar 10.000 conexões simultâneas.

Foi desenvolvido para ser uma arquitetura baseada em fluxo de tarefas, garantindo velocidade de processamento e escalabilidade.

4.2.14 Git

O git é um sistema de controle de versão moderno, sendo o mais utilizado do mercado. É um projeto de código aberto que recebe constantes atualizações de seus contribuidores. Foi desenvolvido em 2005 por Linus Torvalds, o mesmo criador do *kernel* do sistema operacional Linux. O projeto foi construído a fim de desburocratizar o controle de versionamento, visto que, na época o único software de versionamento, o Bitkeeper, passou a ser licenciado e pela revogação do seu uso de forma gratuita.

5 Metodologia

O projeto prático foi desenvolvido com base em reuniões realizadas com policiais do 9º BAEP, junto a eles foram levantados os requisitos do sistema, possibilitando desenvolver a camada da REST API, contemplando todas as regras de negócio mapeadas, sendo validadas pelos cenários de testes produzidos, pelo protótipo desenvolvido no *Figma* e pelos contratos estabelecidos de requisição e resposta da camada *HTTP* entre *back-end* e *front-end*.

Para cada etapa do desenvolvimento do *software*, assim que a tarefa é concluída, ela é submetida em um processo conhecido como *code review*, em que o algoritmo produzido é validado por outro desenvolvedor, este processo colabora com a consistência da aplicação, em caso de erros ou *bugs* encontrados, é necessário que o desenvolvedor do código realize os ajustes necessários. A revisão de código depende de uma ferramenta para o controle de versão do sistema, pois em cada tarefa desenvolvida, aplicou-se as mudanças necessárias no ambiente destino, para tal, foi utilizado o GitLab, que colabora com a visualização da diferença de código, exibindo o que é novo, o que foi removido ou alterado, e permitindo que comentários sejam aplicados nas linhas de código.

O versionamento do sistema foi realizado através do Git e cada uma das *features* foram desenvolvidas em suas respectivas *branches* e submetidas ao GitLab. Cada tarefa foi definida por um escopo *invest*, que se diz respeito ao desenvolvimento de todos os passos necessários para a conclusão de uma determinada ação, concluindo assim um requisito funcional do projeto.

Para a validação de cada tarefa, foi definido um "conceito de pronto", este, estipula alguns parâmetros para o qual a tarefa deve cumprir para ser aprovada e liberada para o uso. Neste projeto as seguintes regras foram acordadas:

- Ao menos 95% de cobertura de código;
- Aprovação de 100% dos testes;
- Respeitar os princípios SOLID;
- Respeitar as normas da PSR-12;
- Passar pela análise estática;
- A *feature* deve ter passado por revisão de código;
- A tarefa deve cumprir o requisito funcional estipulado.

A fim de respeitar o conceito de pronto definido, as regras foram aplicadas em um *pipeline (Continuous Integration)* que é executado sempre que novas mudanças são submetidas.

6 Desenvolvimento

Com base na comunicação entre o policiais do 9° BAEP, foram mapeados os seguintes requisitos:

Requisitos Funcionais

- Controlar as viaturas do batalhão;
- Controlar abordagem de civis;
- Controlar civil procurado;
- Controlar abordagem de veículos;
- Controlar veículos procurados;
- Controlar naturezas aceitas;
- Controlar produtividades aceitas;
- Armazenar informações sobre ocorrências atendidas;
- Permitir que policiais montem equipes;
- Permitir emissão de histórico de ações compartilhadas entre os policiais de uma mesma equipe;

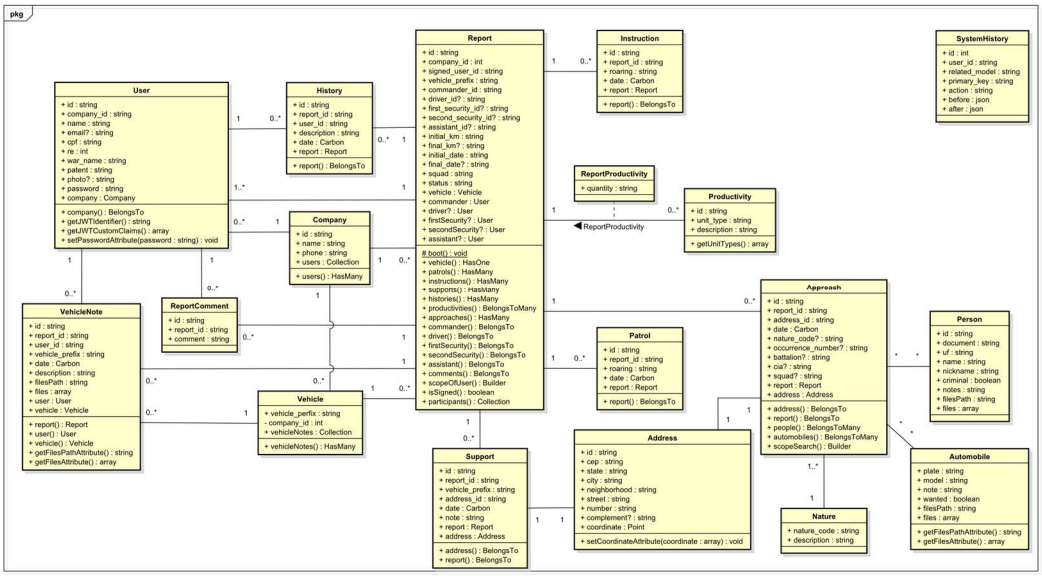
- Permitir visualização dos RSO's buscando por policial e data;
- RSO deve seguir um fluxo para validação (Em andamento, revisão, retornado, assinado);
- Permitir que patentes superiores possam assinar e revisar RSO;
- Emitir o documento RSO em XLSX.

Requisitos não funcionais

- Todas as ações realizadas no sistema devem ser auditáveis não repudiáveis;
- Acesso somente por meio de VPN;
- Sistema WEB desenvolvido em PHP e MYSQL, possibilitando integração com outros sistemas internos;
- Deve funcionar de acordo com os requisitos do servidor do CPI-5.

A partir dos requisitos do projeto definidos, foi realizada a modelagem das entidades do sistema e como elas se relacionam, levando em conta que sistema foi desenvolvido baseando-se no modelo *MVC*, para disponibilizar uma interface de aplicação *REST API*, o que resultou no diagrama de classes (Figura 4). Em suma, há empresas que possuem seus respectivos usuários, eles podem registrar os relatórios de serviço operacional, cada um deles são preenchidos e são completados pelas entidades relacionadas. Sempre que qualquer ação de criação, edição ou deleção for executada, os modelos emitem eventos que registram o histórico (representado pela classe *SystemHistory*).

Figura 4 Diagrama de Classes



Fonte: Autores, 2021.

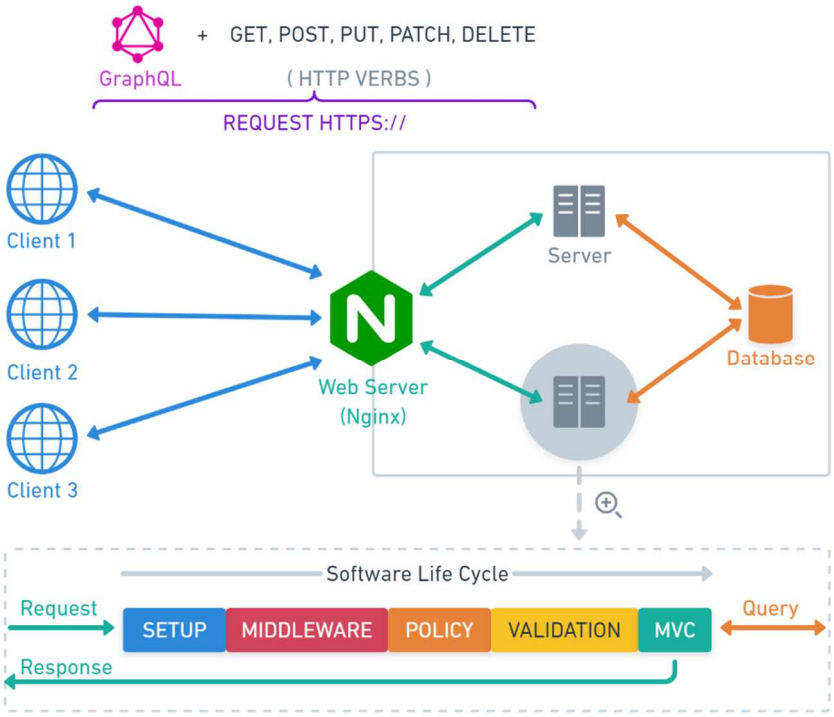
A arquitetura do sistema foi planejada para desenvolver uma *REST API* que pudesse assegurar a escalabilidade do projeto, através do *Nginx*, que possibilita a comunicação entre cliente e servidor de maneira estável, priorizando a performance das requisições. O servidor suporta quaisquer tipos de verbos do protocolo *HTTP* e consultas feitas em *GraphQL*, que utiliza o verbo *POST* para realizar qualquer comunicação com o servidor.

As requisições foram feitas pensando na melhor comunicação entre o *front-end* e o *back-end*, visto que os *CRUD´s* desenvolvidos sempre seguem um padrão. Foram utilizadas as próprias rotas *HTTP* para realizar a interação com o *software*. No entanto, no módulo analítico, dado a complexidade e as possibilidades de realizar diferentes consultas, fizeram com que os dados retornados pelas rotas fossem excessivos ou insuficientes em relação aos dados que precisavam ser manipulados. A fim de que o problema pudesse ser evitado, foi adicionado o *GraphQL*, agindo diretamente na consulta dos dados, diminuindo a complexidade do *back-end* e do tráfego de dados do servidor para o cliente.

Para atender aos requisitos de segurança, toda requisição é submetida a tratamentos especiais, passando primeiro por *middleware* que lidam com a autenticidade da requisição, após este processo, a requisição passa por duas etapas de validação no *form request* correspondente a rota que o usuário está acessando. A primeira etapa foi para validar se o usuário possui a permissão de realizar a ação correspondente através das *policies*, enquanto a segunda etapa foi para validar se os dados submetidos estão sendo validados conforme as regras, passando por *rules* que exigem que a informação esteja devidamente formatada.

Após passar por todas as etapas, a requisição estará pronta para ser executada, e os dados submetidos são enviados para a camada *controller* que irá se responsabilizar por garantir a sequência de ações a serem tomadas, manipulando os dados através das *models*. Em decorrência do tratamento da *model*, eventos podem ser disparados, em *observers* e *traits*, que irão lidar com pré ou pós processamentos da *model*. Ao encerrar o ciclo, a *API* retorna uma resposta em *JSON* para o cliente, informando o sucesso ou a falha da requisição. A arquitetura do sistema está representada na Figura 5.

Figura 5 Arquitetura do Sistema



Fonte: Autores, 2021.

Acordado os padrões de desenvolvimento, os requisitos foram transferidos para tarefas em um *kanban online*, juntamente com as descrições do escopo da tarefa, onde os envolvidos no projeto puderam se organizar para iniciar o desenvolvimento.

O desenvolvimento do código foi baseado seguindo padrões de qualidade, como o *KISS* (*Keep it Simple and Stupid*): um conceito utilizado para sempre lembrar de desenvolver tudo da forma mais simples possível, o *YAGNI* (*You ain't gonna need it*): um princípio que cita não desenvolver algo com a premissa de usá-lo futuramente, o *SOLID*: acrônimo para os seguintes padrões:

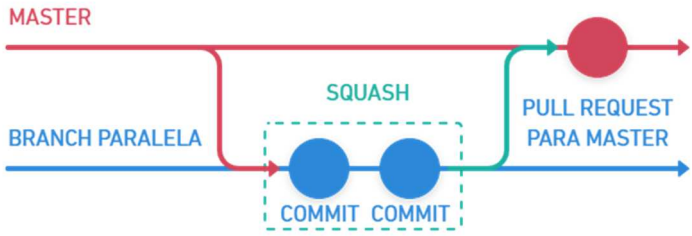
- S - *Single Responsability Principle* (Princípio da responsabilidade única);
- O - *Open-Closed Principle* (Princípio Aberto-Fechado);
- L - *Liskov Substitution Principle* (Prinípio da substituição de Liskov);
- I - *Interface Segregation Principle* (Princípio da segregação da interface);
- D - *Dependency Inversion Principle* (Princípio da inversão da dependência).

Além disso, foram utilizados conceitos de *Object Calisthenics*: regras baseadas no *SOLID* desenvolvidas para garantir uma melhor manutenção, legibilidade, testabilidade e compreensão de código. Suas regras são definidas em 9 princípios:

- *Only One Level Of Indentation Per Method;*
- *Don't Use The Else Keyword*;
- *Wrap All Primitives And Strings;*
- *First Class Collections;*
- *One Dot Per Line;*
- *Don't Abbreviate;*
- *Keep All Entities Small;*
- *No Classes With More Than Two Instance Variables;*
- *No Getters/Setters/Properties.*

Por fim, para o versionamento do código, foi utilizado o *Git*. As tarefas especificadas no quadro *kanban* se convertem em novas *branches* com o prefixo "*feature/nome-do-recurso*" criadas a partir de *master*, como demonstra a Figura 6.

Figura 6 Fluxo de criação de features



Fonte: Autores, 2021.

A disponibilização da *feature* para a *branch master* só é permitida quando a tarefa cumprir todos os parâmetros estipulados no conceito de pronto do projeto.

6.1 Suíte de Testes

O processo de criação dos testes deve ser feito com cautela, pois apenas produzir testes unitários em um algoritmo visando atingir uma cobertura de 100% do código final, não garante que a regra de negócio esteja devidamente testada contemplando todos os possíveis cenários. Como por exemplo no Algoritmo 1, que descreve uma *policy* do sistema, responsável por validar se um usuário pode editar um relatório de serviço operacional, retornando verdadeiro somente se o usuário participou da equipe e o relatório não está assinado.

```
PROGRAMA Validar_Alteração_Relatório;                                    (1)

INÍCIO
       BOOLEANO: participa_equipe, RSO_não_assinado;

       LER participa_equipe, RSO_não_assinado;
       SE participa_equipe E RSO_não_assinado
              RETORNAR VERDADEIRO;
       SENÃO
              RETORNAR FALSO;
       FIM-SE
FIM
```

O Algoritmo 1 resultou na implementação de dois testes unitários, representados pelos Algoritmos 2 e 3.

```
PROGRAMA Verdadeiro_Se_Usuário_Participa_Da_Equipe_E_RSO_Não_Assinado     (2)

INÍCIO
       participa <- VERDADEIRO;
       não_assinado <- VERDADEIRO;
       retorno <- Validar_Alteracao_Relatório participa, não_assinado;
       SE retorno = VERDADEIRO ENTÃO
              ESCREVA "passou no teste";
       FIM-SE
FIM


PROGRAMA Falso_Se_Usuário_Não_Participa_Da_Equipe_E_RSO_Assinado          (3)

INÍCIO
       participa <- FALSO;
       não_assinado <- FALSO;
       retorno <- Validar_Alteracao_Relatorio participa, não_assinado;

       SE retorno = FALSO ENTÃO
              ESCREVA "passou no teste";
       FIM-SE
FIM
```

Com apenas os dois testes, foi alcançado uma cobertura de código de 100%, isso pois somados, resultaram na execução de todas as linhas de comando, entretanto não garantam a imutabilidade da regra de negócio, pois quando se observa a tabela verdade resultante da expressão lógica E (Quadro 2), que representa todos os possíveis resultados do Algoritmo 1, os

cenários testados são os mesmos correspondentes a uma tabela verdade de um operador lógico OU (Quadro 3).

Quadro 2 Tabela Verdade - Operador AND

Contemplado	P	Q	P ^ Q
Sim	V	V	V
Não	V	F	F
Não	F	V	F
Sim	F	F	F

Fonte: Autores, 2021.

Quadro 3 Tabela Verdade - Operador OR

Contemplado	P	Q	P v Q
Sim	V	V	V
Não	V	F	V
Não	F	V	V
Sim	F	F	F

Fonte: Autores, 2021.

O teste de mutação então, irá realizar a modificação do operador lógico nos algoritmos e caso não ocorra quebra de testes, significa que a alteração não teve impacto nos testes, permitindo com que o sistema assuma um comportamento diferente do descrito na regra de negócio, resultando em um relatório informando ao desenvolvedor que aquele trecho de código não está devidamente testado.

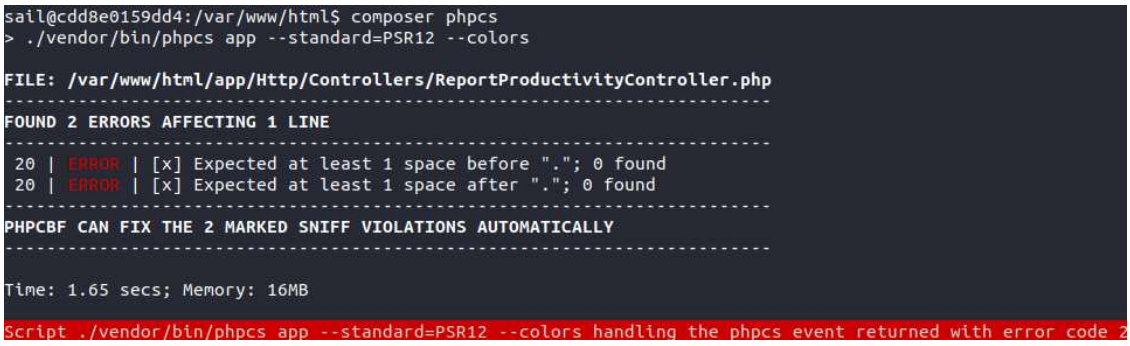
7.2 PHP Metrics

O PHP Metrics foi utilizado para mensurar as métricas de código (incluindo as métricas de Halstead) e entender se as refatorações aplicadas no código através das metodologias de boas práticas colaboraram para melhores resultados.

7.3 PHP Code Sniffer

O pacote “squizlabs/php_codesniffer” foi utilizado para aplicar as regras definidas pela PSR-12 a fim de padronizar a estilização de código em todo o projeto. A figura 7 representa um exemplo de erro encontrado pelo pacote ao perceber que havia a ausência de espaços entre a concatenação de *strings* que é representado pelo símbolo ponto final.

Figura 7 PHP Code Sniffer - Exemplo de violação de sintaxe da PSR-12

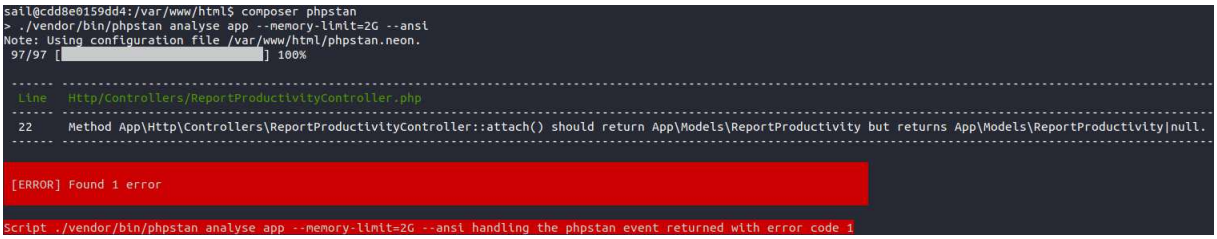


Fonte: Autores, 2021.

7.4 Larastan

O pacote “nonomaduro/larastan” realiza uma validação de análise estática a fim de encontrar possíveis problemas que podem ocorrer com o código ao ser executado, a figura 8 representa um exemplo de erro causado ao informar o retorno de uma função de maneira indevida.

Figura 8 PHP Stan - Exemplo de Erro



Fonte: Autores, 2021.

A ideia principal de utilizá-lo, é validar os erros inspecionados e arrumá-los, garantindo assim que ações indevidas sejam executadas de forma irresponsável. Ao corrigir o problema informado, houve o feedback positivo como demonstra a figura 9.

Figura 9 PHP Stan - Exemplo de Sucesso

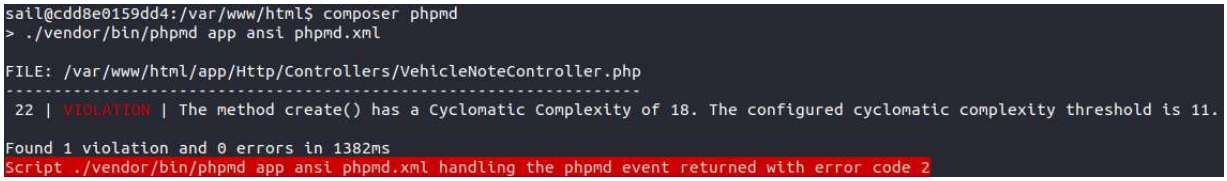
Fonte: Autores, 2021.

Com o auxílio desta ferramenta é possível identificar possíveis erros, de tipagem em uma linguagem fracamente tipada, além de prover um melhor código.

7.5 PHP Mess Detector

O PHP Mess Detector foi utilizado para validar regras aplicadas pelas métricas de Halstead a fim de evitar que o software exceda as métricas convencionadas. A complexidade ciclomática é um exemplo, assim como demonstra a Figura 10, a complexidade ciclomática de uma das controladoras ficou em 18 sendo que o limite era 11, causando uma violação.

Figura 10 PHP Mess Detector - Exemplo de Violação



Fonte: Autores, 2021.

Ao refatorar o código, a complexidade ciclomática do método diminuiu, como demonstra a Figura 11, garantindo que o código esteja dentro das métricas estipuladas.

Figura 11 PHP Mess Detector - Exemplo de Sucesso

Fonte: Autores, 2021.

Com o suporte do pacote, é possível encontrar comportamentos, ou práticas de codificações não concisas, prevenindo possíveis comportamentos indesejados, auxiliando na fase de codificação.

7.6 PHP

O PHP foi utilizado como linguagem de programação devido a necessidade do desenvolvimento de uma aplicação web.

7.6.1 PHP Unit

O PHP Unit foi utilizado para escrever os testes de caixa branca e caixa preta do *software* desenvolvido.

7.6.2 Infection

O *infection* contribuiu para a identificação de locais não testados pelos testes de software, ajudando a aumentar a cobertura de código e evitando possíveis bugs.

7.7 Laravel

O Laravel foi o *framework* escolhido para o desenvolvimento do ambiente *back-end*, utilizado para fornecer uma REST API para o *front-end*.

7.8 GraphQL

O GraphQL colaborou em fornecer possibilidades de *queries* de maneira mais simples para o *front-end,* além de contribuir para diminuir a quantidade de dados retornados da API.

7.9 MySql

O MySql foi utilizado como banco de dados principal para guardar os dados em ambiente de produção e para executar os testes automatizados.

7.10 Docker

O Docker foi utilizado para criar um ambiente de desenvolvimento que pudesse ser único através da conteinerização, além de orquestrar os containers que são utilizados no ambiente de produção.

7.11 Nginx

O Nginx foi utilizado como uma forma intermediária para realizar as requisições para o servidor, isso pois ele pode atuar como um servidor web, fazendo o balanceamento das requisições e lidar com múltiplos servidores caso seja necessário escalar o projeto.

7.12 Git

A fim de realizar o processo de versionamento do projeto, o git foi utilizado para realizar o versionamento do projeto.

Resultados e Discussões

O processo de análise e desenvolvimento de sistemas é algo complexo, indo muito além da simples produção de códigos, para entender o que e como será desenvolvido, partindo das primeiras interações com o cliente, entendendo e mapeando os processos e regras de negócio, levantando os requisitos, acordando padrões de arquitetura e código atendendo às necessidades, criar os cenários de testes, produzir o código, aprovar o que foi desenvolvido e garantir que o que foi produzido está tendo os comportamentos esperados.

Para o desenvolvimento deste projeto foram mapeados 14 requisitos funcionais, decodificados em 436 testes de *software* condizente a cerca de 970 asserções, indicando que cada teste válida no mínimo 2 parâmetros para ser aceito, com uma cobertura de 95% do código, validados com ajuda de testes de mutabilidade, em um sistema *REST API* com 65 rotas produzidos com a ajuda do *framework Laravel*, Com ajuda do *PHP Metrics* foi constatado que a complexidade ciclomática média por classe é de 1,74 em 95 classes, apenas 1 violação que se diz respeito a classe de *Report*, que quebrou o conceito de *god classs*, o que faz sentido visto que o propósito desta classe é unir todos os dados do relatório de serviço operacional.

A aplicação de conceitos que visam garantir um código legível como *SOLID*, *KISS* e *Object Calisthenics* contribuem diretamente com a indicação de boas métricas como a média de 8 linhas de código por método, o que representa em sua maioria métodos pequenos, melhorando o entendimento e manutenção.

Outras métricas importantes atingidas estão descritas na tabela 1, todas dentro dos valores esperados.

Tabela 1: Resultados atingidos.

Métrica	Valores atingidos	Valores esperados
Complexidade ciclomática (CC)	1,74	CC <= 10
Volume (V)	525.14	V <= 800
Difficulty (D)	1,56	D <= 4
Effort to implement (E)	819.2	E <= 1000
Errors (B)	0,17	B <= 5.0

Fonte: Autores, 2021.

Ainda que o *software* fosse perfeito, computadores executam ordens dadas pelos desenvolvedores, desenvolvedores humanos que também cometem erros. Para tornar a taxa de erros cada vez menores existem os testes que validam cada um dos cenários, e processos de revisão de código, mas brechas ainda existem, e podem passar despercebidas com muita facilidade. Para evitar que essas brechas causem danos significativos e para evitar situações indesejadas ocorram, o sistema sempre trabalha com a veracidade. Ainda que um usuário possa fazer um ato ilegal passando pelo *middleware* e pela *policy* o sistema registra *logs* de todas as ações feitas, seja ações de sucesso ou erros, gravando dados como identificador do usuário autenticado, *IP* de acesso, ação feita, horário e o que foi feito. Dessa forma, é possível garantir um dos pilares da segurança da informação, a irritabilidade, não sendo possível contestar os acontecimentos que o sistema registra.

Conclusões

Com a premissa desenvolver uma aplicação back-end que seguisse os requisitos levantados foram aplicadas técnicas de engenharia de software com o intuito de avaliar sua qualidade através de métricas de código que comprovem sua legibilidade, integridade, manutenibilidade e complexidade, refletindo em uma análise dos resultados ao programar em conjunto com boas práticas de programação, foi utilizado as métricas de Halstead juntamente com os pacotes desenvolvidos pela comunidade PHP a fim de auxiliar no desenvolvimento do *software*.

A partir da definição dos requisitos e com o propósito de garantir o funcionamento correto da aplicação, foram desenvolvidos testes de *software* que garantiram através das asserções a integridade do projeto, 100% de cobertura de código pelos testes de *feature* e unitários, além aumentar a eficácia dos testes através do *MSI* com o auxílio do pacote *infection*.

Através das metodologias de desenvolvimento utilizadas: SOLID, KISS, YAGNI e *Object Calisthenics*, foi possível garantir uma melhor legibilidade do código, deixando-o mais simples de realizar manutenções, ao aplicar regras de análise estática fornecidos pelos pacotes PHP Code Sniffer, PHP *Mess Detector* e PHP *Stan*, foi possível garantir maior estabilidade e clareza no desenvolvimento do software. Além disso, o PHP Metrics colaborou com a visualização das métricas de Halstead, contribuindo com a compreensão da dimensão do projeto.

Em decorrência do âmbito do projeto, alguns dos pilares da segurança da informação foram levados em conta no projeto durante o desenvolvimento, logs e histórico de ações são registrados como forma de garantir a irretratabilidade dos fatos, além de técnicas para validar a autenticidade e o permissionamento através dos *middlewares* e das *policies.*

Com o intuito de aprimorar o software, sugerimos o uso de um banco de dados cronológico para lidar com os registros de *logs* e históricos e um banco de dados em memória para lidar com processamento de dados em *background* (*cache*), além de melhorar a conteinerização do projeto através do docker swarm com a finalidade de realizar operações de *deploy zero down-time* (sem que o servidor precise ser desativado) e para garantir que o servidor seja capaz de se restaurar por conta própria a partir de uma queda inesperada.

Por fim, com a unificação dos dados de em uma mesma base, operando em um banco de dados relacional, traz vantagens a BAEP, sendo possível realizar consultas mais precisas, geradas através de dados operacionais, por meio de consultas SQL, por exemplo, é possível identificar valores quantitativos dos tipos de ocorrências, em registros de data e endereço, auxiliando na tomada de decisões estratégicas, podendo assim contribuir com a segurança regional, melhorando a qualidade de vida de nossa sociedade.

Agradecimentos

Ao Capitão Táparo por disponibilizar a oportunidade de desenvolver o projeto e sua colaboração com as dúvidas da equipe; Ao Sargento Oliveira por auxiliar no processo de *deploy* do projeto; aos orientadores, pelo tempo dedicado ao acompanhamento do desenvolvimento do projeto e pelas correções que garantiram o melhor desempenho da equipe; aos professores da Fatec Rio Preto, por terem contribuído para a formação de toda a base acadêmica.

Referências

AJALA, Vinícius; SILVA, Denilson; SANTOS, Cristina; ROLIM, Carlos. **Análise da Complexidade Ciclomática como Apoio ao Processo de Desenvolvimento do Pensamento Algorítmico**. Universidade Regional Integrada do Alto Uruguai e das Missões (URI), 2016. Disponível em: https://ebooks.pucrs.br/edipucrs/anais/csbc/assets/2016/desafie!/04.pdf. Acesso em: 18 nov. 2021.

BAHIA, Alvaro. Métricas de Halstead. In: BAHIA, Alvaro. **O uso de métricas de complexidade para o controle da qualidade de software científico**. 1992. Tese (Mestrado em ciências em engenharia de sistemas e computação) - Universidade Federal do Rio de Janeiro, [S. l.], 1992. Disponível em: https://www.cos.ufrj.br/uploadfile/publicacao/1463.pdf. Acesso em: 7 dez. 2021.

BERGMANN, Sebastian. PHP Unit - Documentação. [S. l.], 2001. Disponível em: https://phpunit.readthedocs.io/pt_BR/latest/. Acesso em: 17 nov. 2021.

CRISCUOLO, Marcelo. **Qualidade de Produto de Software: uma abordagem baseada no controle da complexidade**. USP São Carlos: Profa. Dra. Rosely Sanches, fevereiro 2008. Disponível em: https://teses.usp.br/teses/disponiveis/55/55134/tde-09052008-145857/publico/dismarcelo.pdf. Acesso em: 18 nov. 2021.

SANTOS, Emanuel. Procedimento de avaliação. In: SANTOS, Emanuel. **Uma proposta de métricas para avaliar modelos**. 2008. Dissertação de mestrado (Mestrado) - Universidade Federal de pernambuco, [S. l.], 2008.

FOWLER, Martin. **Refatoração**: Aperfeiçoando o Design de Códigos Existentes. 2. ed. rev. e atual. [S. l.]: Novatec Editora, 30 de abril 2020. 456 p. ISBN 8575227246.

GITHUB CONTRIBUITORS. **PHP Code Sniffer**. [S. l.], 2021. Disponível em: https://github.com/squizlabs/PHP_CodeSniffer. Acesso em: 17 nov. 2021.

GITHUB CONTRIBUITORS. **PHP Metrics**. [S. l.], 2021. Disponível em: https://phpmetrics.org/index.html. Acesso em: 17 nov. 2021.

GITHUB SPONSORS. **Infection PHP**. [S. l.], 2001. Disponível em: https://infection.github.io/guide/. Acesso em: 17 nov. 2021.

GRAPHQL. [S. l.], 2021. Disponível em: https://graphql.org/. Acesso em: 17 nov. 2021.

JUNIOR, Heleno; PRADO, Alisson; ARAÚJO, Marco. **Complexity Tool: Uma Ferramenta para Medir Complexidade Ciclomática de Métodos Java**. ResearchGate, janeiro 2016. Disponível em:

https://www.researchgate.net/publication/327681295_Complexity_Tool_Uma_Ferramenta_para_Medir_Complexidade_Ciclomatica_de_Metodos_Java. Acesso em: 18 nov. 2021.

LARAVEL. [S. l.], 2021. Disponível em: https://laravel.com/docs/8.x. Acesso em: 17 nov. 2021.

MICROSOFT. **Tutorial: Create a web API with ASP.NET Core**. [S. l.], 12 jul. 2021. Disponível em: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio. Acesso em: 8 dez. 2021.

NETO, Arilo. Introdução a Teste de Software. **Engenharia de Software Magazine**, [S. l.], n. 1, p. 54-59, 14 maio 2015. Disponível em:

MOST Popular Backend Frameworks 2011/2021. Produção: Statistics and data. YouTube: [s. n.], 2021. Disponível em: https://www.youtube.com/watch?v=sFA0mOS7hN4. Acesso em: 8 dez. 2021.

PICHLER, Manuel. **PHP Mess Detector**. [S. l.], 2021. Disponível em: https://phpmd.org/. Acesso em: 17 nov. 2021.

RED HAT. **O que é API REST?**. [S. l.], 8 maio 2020. Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api. Acesso em: 8 dez. 2021.

ROCK CONTENT. **Entenda o que é um Log File e aprenda como criar um**. [S. l.], 12 jun. 2019. Disponível em: https://rockcontent.com/br/blog/log-file/. Acesso em: 8 dez. 2021.

SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 5., 2006, Universidade Federal de Pernambuco. **Um Processo de Análise de Cobertura alinhado ao Processo de Desenvolvimento de Software em Aplicações Embarcadas** [...]. [S. l.: s. n.], 2006. Disponível em:
https://d1wqtxts1xzle7.cloudfront.net/42778565/Artigo_Um_Processo_de_Analise_de_Cobertura.pdf?1455742080=&response-content-disposition=inline%3B+filename%3DUm_Processo_de_Analise_de_Cobertura_alin.pdf&Expires=1639029117&Signature=Z6sfqOQbtY5FmBsmtyvboq~rp-z096ZsPbZPeupimo42mA~RlZm0Ghceq-UuQvW55G5jaORNq0WZs~k3EuW3WianEnzDvkL7KpbUf1QRz2ACTvOU-r6m2CkIEbsFyAfboAlU6pe2O~V1BnnFzq2IPT-UD~NoHdyZzauAc~kZgHvvY~2BfGWdsQ9SEh~TAeLH5Cb2isRydo1eR4fRTW6TThogbIU98AjKrXcnu1wP1ziHNjo2MiBVuiJs-GesJoy5FpuiWqKCklAmUfUxqZ2mwe0HAnnOo9q-

K7j~Sj7beNO~cSqxEm6cikJLQu8Dq3WrSKr4LoB37AfNvzaCz2jpLw__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA. Acesso em: 8 dez. 2021.

TREINA WEB; GUEDES, Marylene. **O que é MVC?**. [S. l.], 2020. Disponível em: https://www.treinaweb.com.br/blog/o-que-e-mvc. Acesso em: 8 dez. 2021.

https://edisciplinas.usp.br/pluginfile.php/3503764/mod_resource/content/3/Introducao_a_Teste_de_Software.pdf. Acesso em: 8 dez. 2021.

W3SCHOOLS. ***History Of Laravel***. [S. l.], c2021. Disponível em: https://www.w3schools.in/laravel-tutorial/history/. Acesso em: 8 dez. 2021.