Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Biblioteca de geração automática de testes de regressão a partir do log de execução de aplicações SaaS

Frederico Dib

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Genaína Rodrigues

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Biblioteca de geração automática de testes de regressão a partir do log de execução de aplicações SaaS

Frederico Dib

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Genaína Rodrigues (Orientador)
CIC/UnB

do Bacharelado em Ciência da Computação

Brasília, 01 de fevereiro de 2021

Dedicatória

Dedico esse trabalho aos meus amigos da UNB e aos meus pais que sempre me deram apoio e me ajudaram quando necessário.

Agradecimentos

Agradeço aos meus pais pelo apoio nos estudos e agradeço em especial à mentora Vanessa Tavares Nunes e à Prof.a Dr.a Genaína Nunes Rodrigues, a professora orientadora que tornou esse trabalho possível.

Resumo

Atualmente a complexidade dos sistemas de software vem crescendo muito, o que torna um grande desafio para a entrega do software em um tempo curto. A competividade e a necessidade de inovação tem demandado sucessivos lançamentos de novas versões de seus softwares em um ritmo cada vez mais acelerado.

Como forma de manter a qualidade destes softwares e evitar erros despercebidos ao lançar nova versão, podem ser utilizados testes automáticos, capazes de testarem um sistema completo em questão de minutos, evitando-se, dessa forma, possíveis *bugs* ou pior, problemas graves.

No entanto, apesar da importância de testes de softwares já serem um tema reconhecido em sua importância, existe uma negligência muito grande em seu uso. Em particular, no caso de startups e pequenas empresas, que geralmente não possuem recursos financeiros e contam com uma equipe muito enxuta, tendem a dar preferência ao lançamento das novas funcionalidades de negócios do que construir um código mais sólido e robusto.

Visando dar condições para que as startups consigam agilizar e aprimorar a capacidade de testes de software, este trabalho se propõe a criar uma gema em *Ruby*, para ser rodada em *Ruby on Rails API*, que gere automaticamente testes de regressão que simulam o comportamento do uso e dos testes manuais realizados em um sistema.

Desta maneira, esta ferramenta possibilitaria que o teste manual só necessitasse ser realizado uma única vez, persistindo na forma de testes automatizados em formatos de testes de regressão para a funcionalidade testada. Após a geração do teste automatizado não seria necessário repetir os testes manuais novamente, aumentando potencialmente a velocidade de desenvolvimento e estabilidade do projeto.

Palavras-chave: Behavior, Behavior Driven Development, Testes, Engenharia de Software, Testes Unitários, Automação de testes

Abstract

Currently, the complexity of systems, in the area of software development, has been growing a lot, while the time for development is getting shorter and shorter. The competitiveness and innovation of the business universe on the web, generally, has demanded successive launches of new versions of its software.

As a way of maintaining the quality of these software and avoiding unnoticed errors when launching a new version, automatic tests can be used, capable of testing a complete system in a matter of minutes, thus avoiding possible *bugs* or worse, serious problems. Despite the importance of software testing is already a pacified topic, there is a great deal of neglect in its use. In the case of startups and small companies, which generally do not have financial resources and have a very lean team, they tend to give preference to the launch of new business features than to build a more solid and robust code.

Aiming to provide conditions for startups to be able to streamline and improve the capacity of software testing, this work proposes to create a gem in *Ruby*, to be run in *Ruby on Rails API*, which automatically generates regression tests that simulate the behavior of use and manual tests performed on a system.

In this way, this tool would make it possible that the manual test only needed to be carried out once, generating automated tests for tested functionality. After generating the automated test, it would not be necessary to repeat the manual tests again, thus increasing the speed of development and stability of the project.

Keywords: Behavior, Behavior Driven Development, Testing, Software Engineering, Unit Test, Automated test generation

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A ação de se testar um software, seja de forma manual ou automatizada, é um processo fundamental e necessário para evitar erros e/ou problemas de funcionamento. E com o crescimento do tamanho e da complexidade dos softwares nos últimos anos os testes manuais se tornaram cada vez mais ineficientes. A realização de testes manuais é uma atividades demorada e recorrente, de maneira que a cada mudança, muitas vezes acaba se tornando necessária a realização de todos os testes manuais. Além disso estes podem não possuir uma grande cobertura em relação ao verificar e validar o código de todas as formas possíveis, permitindo, muitas vezes, que problemas e funcionamentos indevidos passem despercebidos. Segundo [1] "alguns desses erros não são importantes, mas alguns deles são caros e perigosos".

Os testes automatizados, segundo [2], por estarem menos suscetíveis a erros humanos e por serem mais escaláveis, são a forma mais eficiente de se testar um software. Esses testes significam utilizar um agente (um sistema ou componente de sistema) computacional para verificar se o comportamento de um software ou o comportamento de um componente está de acordo com o esperado, com base em um conjunto de entradas específicas. Esses testes podem ser classificados de diferentes maneiras, sendo os testes de comportamento, os testes de caixa preta os tipos que abordaremos.

Com a popularização das metodologias ágeis, os testes de softwares também seguiram as regras do manifesto ágil, pois se encaixavam muito bem, devido ao modelo de sprints. Cada sprint poderia ser testada antes da definição da sprint seguinte. A metodologia Ágil foi criada devido a necessidade de tornar o processo de desenvolvimento mais dinâmico, dessa maneira, garantindo que o cliente recebesse um software que realmente satisfizesse suas necessidades, com maior qualidade e com uma entrega mais rápida, do que pela metodologia antecedente (cascata)[3]. Essa metodologia se baseia no conceito de iterações e de sprints, sendo sua duração predefinida pela equipe, que valida com o cliente o que foi implementado e definem as principais prioridades a serem iteradas no próximo ciclo,

de acordo com algum critério de priorização. Pelo projeto estar sempre em constantes iterações e mudanças, os testes automatizados de software, dão uma maior segurança e agilidade, garantindo que nenhuma nova iteração irá prejudicar uma iteração antiga.

1.1 Problema

Apesar da importância dos testes automatizados e da vantagem de só precisarem serem feitos uma única vez, para sua criação e efetividade é necessário um trabalho desafiador de implementar um número significante de testes e de garantir que eles englobem todos os requisitos predefinidos. Sendo assim, em termos de trabalho, a construção desses testes pode onerar demais as empresas, fazendo com que eles sejam negligenciados por empresas menores, principalmente startups.

Startups são pequenas empresas que buscam, a partir de um problema de mercado, criar soluções escaláveis que resolvam esses problemas. Por natureza, startups são negócios de alto risco e baixa taxa de sucesso. Para diminuir esses riscos elas possuem uma cultura de buscar validar seu produto e solução da forma mais simples e rápida possível, para aprender, gerar dados, e testar uma nova variação de seu produto, até por fim, quando tiver insumos o suficiente, seguir com o produto e o modelo de negócio pelo caminho mais promissor, de forma que, em muitos casos, o resultado é bem diferente do modelo inicial planejado

Tendo a cultura das startups em mente, observamos que a qualidade do software não é vista como prioridade, sendo negligenciada até se obter um produto mais maduro [4]. Portanto, a situação mais percebida são startups negligenciando os testes automatizados e optando pelos manuais apenas, podendo gerar dificuldades e problemas a partir do crescimento dos softwares e mudanças de funcionalidades.

1.2 Objetivo geral

Este trabalho propõe a criação de uma biblioteca (gema) a ser usada dentro do framework Ruby on Rails Backend que a partir do teste manual do software gere automaticamente testes de de regressão automatizados a partir do log de execução de aplicações SaaS.

Apesar de ainda ser necessário realizar o teste manual para a utilização da biblioteca, o principal ganho ao utilizá-la é o fato de ser necessário realizar um teste manual apenas uma única vez. Ao finalizar uma *sprint* e testar manualmente a feature recém criada, são gerados os testes de regressão. Após finalizar outra sprint, não será necessário testar manualmente novamente as funcionalidades anteriores, pois os testes automatizados da mesma já terem sido realizados.

1.3 Objetivos específicos

O objetivo específico desse trabalho é construir uma gema de Ruby [5] para ser usada em aplicações Ruby on Rails [6] como uma aplicação de API (Application Programming Interface) [7] que é uma aplicação de backend exclusivamente. Assim, tendo a finalidade de transformar testes manuais, feitos em ambiente de desenvolvimento, em testes de regressão automatizados. Neste trabalho abordaremos os seguintes itens:

- Criação de uma metodologia de geração de testes de regressão a partir de rastros de execuções em uma arquitetura MVC (Model View Controller).
- Construção da instalação e da inicialização da gema de geração automática de testes de regressão.
- Prova de conceito para evidenciar a eficácia da gema.

1.4 Organização do trabalho

Os demais capítulos desta monografia estão organizados da seguinte forma:

- Capítulo 2: Fundamentos Teóricos. Nesse capítulo será apresentada toda a base teórica utilizada neste trabalho.
- Capítulo 3: Visão Geral da Proposta de Solução. Nesse capítulo é apresentada a ideia geral por trás da implementação da gema, sem especificar uma tecnologia em específico.
- Capítulo 4: Acervo de Bibliotecas. Nesse capítulo são apresentadas todas as bibliotecas e ferramentas utilizadas na implementação da gema.
- Capítulo 5: Implementação do Projeto. Nesse capítulo será apresentado o passo a passo na implementação da gema, como instalá-la e um exemplo de uso.
- Capítulo 6: Casos de Uso. Nesse capítulo será apresentada a cobertura de testes alcançada, assim como outras características, pela gema em alguns sistemas simples.
- Capítulo 7: Conclusão. Nesse capítulo concluiremos esse trabalho, e discutiremos alguns trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Neste capítulo serão apresentadas as principais bases teóricas que foram usadas no desenvolvimento da biblioteca e no embasamento teórico necessário para entender sua necessidade.

2.1 Metodologia Ágil

A metodologia ágil [8] surgiu por volta de 1990 como uma alternativa à metodologia cascata, que embora funcione bem para sistemas pesados e com requisitos que oscilam pouco, para sistemas que estão sempre em mudanças possuindo uma dinamicidade alta, era uma metodologia muito burocrática e lenta. A metodologia Cascata tem como princípio documentar todo o escopo e o passo a passo do projeto antes de iniciar a codificação, de maneira que a documentação deve ser seguida com muito rigor. Nesse contexto, esse procedimento possuía muitas falhas, em muitos momentos, durante a implementação do código, os desenvolvedores se deparavam com problemas que não foram previstos anteriormente, resultando em atrasos. E mesmo ao final do projeto, nem sempre o resultado era o que o cliente estava esperando.

Para solucionar esses problemas surgiu a Metodologia Ágil, que de acordo com o manifesto ágil [9] "Indivíduos e interações mais que processos e ferramentas", sendo essa uma visão em coloca o usuário no centro do desenvolvimento. Na metodologia ágil o projeto é dividido em vários projetos menores conhecidos como Sprint. A cada Sprint é definido o próximo passo a ser implementado, é realizada a documentação, desenvolvida a funcionalidade, e por fim, ao final do ciclo, que costuma durar entre uma a duas semanas, é validada com o cliente o atendimento as suas expectativas e necessidades, e após isso a equipe segue para a próxima Sprint.

Essa natureza incremental do desenvolvimento ágil, casou-se muito bem com a automatização de testes, que estava em ascensão na mesma época. Em paralelo com o

desenvolvimento de uma funcionalidade, os testes automatizados são feitos, dando uma maior segurança não apenas do correto funcionamento da funcionalidade, como também, ao se incrementar uma nova funcionalidade, garantir que as anteriores ou já existentes continuam funcionando. Acelerando assim a velocidade de desenvolvimento.

2.2 Testes de Software

Testes de software fazem parte do processo de desenvolvimento de um software, tendo como principal objetivo encontrar e revelar bugs e falhas no funcionamento do sistema, para que sejam corrigidas. Atualmente existem muitos tipos diferentes de testes, e apesar de cada um ter seus prós e contras, eles não são excludentes entre si, podendo ser adotados vários tipos de testes em um projeto de desenvolvimento de software. Neste trabalho falaremos sobre alguns desses testes:

2.2.1 Testes Unitários

Os testes unitários são testes automatizados que tem como objetivo testar pequenas partes de um sistema individualmente. Esses testes costumam ser feitos em funções, métodos e classes a partir de um conjunto de entradas possíveis e analisando a saída esperada.

Testes unitários são a base de uso do TDD (Test-Driven Development), uma metodologia bastante popular que se consiste em construir os testes de um determinado fragmento de código antes do próprio fragmento de código.

A maior vantagem desses testes é que são testes geralmente mais simples e rápidos de serem feitos e com um rápido tempo de execução também, permitindo testar grandes variedades de entradas em uma função em muito pouco tempo. O ponto negativo desses testes é que para serem construídos é necessário ter um conhecimento específico sobre como deve funcionar cada fragmento de código, e isso se torna difícil de ser realizado pelos desenvolvedores em softwares legados que não foram bem documentados e estruturados.

2.2.2 Testes Manuais

Os Testes Manuais de software, assim como o nome diz, são o ato de um humano testar o comportamento de um software para ver se funcionalidades atuam conforme o esperado. Estes testes podem tanto ser feitos em pequenas unidades do código quanto no comportamento final do sistema. As maiores vantagens desse tipo de teste é que, por não exigir codificação, se torna o tipo de teste mais rápido de ser executado, e, por ser mais dinâmico, muitas vezes permite aos desenvolvedores identificar falhas que muitas vezes não seriam descobertas por testes automatizados. Porém os testes manuais por serem

testes não persistentes, a cada nova iteração no sistema é necessário ser completamente refeito, tornando meno eficiente a médio e longo prazo que os testes automatizados, devido ao grande número de repetições e crescente número de funcionalidades a serem testadas, principalmente considerando sistemas mais robustos. Outro grande problema é que por serem testes executados por humanos, são mais suscetíveis a erros como desatenção, erros de digitação, entre outros.

2.2.3 Testes Funcionais

Os testes Funcionais, ao contrário dos testes Unitários, buscam testar se o comportamento do software como um todo está de acordo com o esperado. Esses testes muitas vezes são testes de caixa preta, o que significa que não é analisada a forma como o código foi organizado e modularizado, sendo apenas visto se a integração de todos esses módulos estão funcionando de forma correta, a partir de uma ação e o resultado final esperado.

Uma das grandes vantagens desses testes é que eles testam o cenário mais próximo da realidade possível, e também são ótimas opções para testar softwares legados em que os desenvolvedores sabem o resultado esperado, mas não possuem total clareza do funcionamento de todo os fragmentos de códigos. Porém esse tipo de teste geralmente é muito demorado para se codificar e de grande complexidade em sua implementação, principalmente quando o software faz acesso a serviços externos, como por exemplo um serviço de pagamento ou alguma API da google, entre outros tipos de integrações.

O teste de comportamento pode ser tanto utilizado em softwares fullstack, que são softwares que tanto o backend quanto o frontend são tratados no mesmo lugar, quanto em softwares Backend API, que são softwares desacoplados do frontend, se comunicando por meio de requisições http e retornando um arquivo de dados, como por exemplo, um arquivo json [10]. Um exemplo de um cenário de teste de comportamento em um software fullstack e um API que visam testar um cenário que trata de visualizar um item de determinada loja poderia ser escrito da sequinte forma:

Software Fullstack

```
Given I am in the catalog page
When I click in "show more" button
Then I see the item information
```

Software Backend API

```
Given there is a item with id 22
When I request with method GET to the url https://exemple_host/items/22
```

```
3 Then I receive a JSON file with the informations of the item
```

Neste trabalho abordaremos principalmente o teste em um contexto de Backend API, o qual possui sua estrutura descrita no próximo capítulo. Além da estrutura de teste é importante que esses testes sejam escritos com qualidades e garantam uma cobertura sobre o código, como falaremos na próxima sessão.

2.2.4 Teste de Regressão

Ao longo da construção da biblioteca, foram utilizados testes de regressão. Este tipo de teste é definido como o processo de repetir os testes de partes modificadas do software, com o objetivo de assegurar que estas modificações não implicaram no acréscimo de novos erros em partes já testadas [11]

Os testes de regressão costumam envolver a reutilização de testes projetados para a primeira versão de um programa em uma versão modificada deste mesmo programa. A partir disso, é possível fazer uma análise e determinar onde os novos casos de teste podem ser necessários para testar desta vez a funcionalidade adicionada após as modificações [11]

Aprofundando-se no conceito introduzido acima, existem diferentes técnicas de aplicação dos testes supramencionados. A partir do estudo dos testes de regressão, destacam-se quatro técnicas possíveis: reteste completo, seleção de teste de regressão, priorização de casos de teste e a abordagem híbrida [11].

A primeira abordagem, como a própria nomenclatura sugere, inclui a aplicação de testes de regressão em todos os testes existentes no conjunto, executando-os novamente [11]. Em contraste, a seleção de testes de regressão envolve uma seleção de parte do conjunto de testes existentes. Para a seleção, os testes existentes são divididos entre casos reutilizáveis, casos retestáveis e casos obsoletos de teste [11].

O terceiro método, de priorização de casos de teste, consiste prioriza determinados casos de teste com o objetivo de otimizar a detecção de falhas [11]. Este tipo de aplicação pode seguir a linha de priorização geral, que tent selecionar uma ordem que sirva a versões futuras do software ou uma prioriação específica à versão, que se atém a uma versão particular [11].

Por último, a abordagem híbrida envolve ambas as técnicas de seleção de casos de teste e priorização [11]

Neste trabalho o conceito de testes de regressão é utilizado na implementação da biblioteca, de maneira que ela transforme todos os testes manuais do desenvolvedor em testes de regressão. Após se implementar uma nova funcionalidade todos os testes gerados anteriormente são re executados para garantir que o sistema mantém seu funcionamento.

2.3 Qualidade de teste

Existem muitas formas de construir testes automatizados. Surge então uma dúvida: o que é necessário para afirmar se o teste construído possui ou não uma boa qualidade? Armando Fox e David Patterson [12] defendem que um teste que possui uma boa qualidade e proporciona uma boa manutenibilidade deve seguir o principio FIRST, em que cada letra possui um significado:

- **F**ast: Deve ser fácil, rápido e simples executar um conjunto de testes referente a uma funcionalidade desenvolvida. Testes muito longos para serem executados podem interferir na linha de pensamento.
- **I**ndependent: Os testes devem ser independentes entre si. Em outras palavras, a ordem que os testes são executados, se um novo foi adicionado, ou algum antigo for removido, essas mudanças não devem afetar os testes existentes.
- **R**epeatable: Os testes não devem depender de fatores externos, com data do dia, ou constantes que podem ser alteradas com frequência. Um problema do gênero ocorrido foi o caso "Y2K problem", em que muitos sistemas da década de 60 pararam de funcionar com a chegada do ano 2000.
- **S**elf-checking: Cada teste deve determinar por si só se ele falhou ou passou, sem depender que um humano avalie o resultado para concluir se existe alguma falha no sistema.
- **T**imely: Os testes devem ser criados ou atualizados concomitante a criação do código. No TDD por exemplo, os testes são criados antes mesmo do código por eles descritos.

Os testes gerados pela biblioteca descrita neste trabalho devem seguir o principio FIRST para garantir sua eficiência. Existe uma pequena ressalva no ponto "Timely", pois como nossos testes serão gerados a partir do uso manual do código, eles apenas serão criados após a finalização do código. Nos capítulos posteriores serão abordados com maior detalhamento como eles são construídos e seu funcionamento.

2.4 Cobertura de teste

Quando falamos sobre testes, uma dúvida frequente é quanto teste é necessário ser feito para garantir com um grau de certeza que o código está funcionando. Não existe uma forma de se garantir que o código funciona conforme o esperado em todas as situações, mas como podemos ver no livro [12] existem algumas formas de se metrificar essa certeza.

Uma métrica que foi bastante utilizada é a razão de linhas de código dividida pela quantidade de linhas de testes. Geralmente essa métrica é deve ser mantida em menos do que 1, significando que existem mais linhas de teste que linhas de código. Apesar de ainda ser muito utilizada, essa métrica está muito sujeita a falhas, portante a métrica mais moderna, que tem sido mais utilizada no momento é a métrica de cobertura de código. Essa métrica basicamente analisa se para cada trecho de código existe um trecho de teste associado a ele. Seguindo [12] podemos classificar os seguintes tipos de cobertura:

- Cobertura em nível de Métodos: Verifica se cada método é executado ao menos uma vez.
- Cobertura em nível de Chamadas: Verifica se cada método é chamado em todos os locais que poderia ser chamado. Como por exemplo um método que chama outro método.
- Cobertura em nível de Declarações: Verifica se cada declaração de código é usada durante a execução dos testes.
- Cobertura em nível de ramos: Verifica se cada ramificação do código é chamada em diferentes cenários. Como por exemplo o caso de uso de condicionais (por exempo, if then else).
- Cobertura em nível de Caminho: Verifica se foram executados todos os caminhos e percursos possíveis através do código. Em outras palavras, se todas as linhas de código foram acessadas em algum momento.

Nesse projeto analisaremos a cobertura a nível de Caminho, ou seja, se todas as frações de código foram acessadas pelo teste. Dessa maneira será atribuído um valor em porcentagem associado a esse proporção. Por exemplo, 100% de cobertura significaria que todas as bifurcações e linhas de código estão sendo levadas em conta nos testes. Como dito no início, a cobertura de código não garante que não há falhas no código, mas atribui um grau de certeza que dentre os cenários pensados, o código está funcionando corretamente.

2.5 Conceitos de desenvolvimento

Nesta sessão abordaremos alguns temas importantes para o completo entendimento da solução proposta nesse trabalho, sendo eles: A Programação Orientada a Objeto é a paradigma base na linguagem Ruby [13] e no framework Ruby on Rails [6] que serão abordados no Capítulo 4. A Programação Orientada a Aspectos e os Ambientes de desenvolvimento

comumente utilizados, são necessários para entender o funcionamento da biblioteca junto de suas limitações, sendo essas apresentadas no Capítulo 3 deste trabalho.

2.5.1 Programação Orientada a Objeto

Programação Orientada a Objeto, também conhecida como POO [14], surgiu nos anos de 1967 pela linguagem "Simula 67", pelos criadores Kristen Nygaard e Ole-Johan Dahl no Centro Norueguês de Computação em Oslo. E desde então a prática se popularizou muito, sendo hoje o paradigma de computação mais utilizado.

A Programação Orientada a Objeto se baseia no conceito de classes e objetos, em que os objetos são instâncias das classes e as classes são escopos que definem como deve ser o objeto, possuindo assim, atributos que são as características e informações guardada por um objeto e métodos que são as ações que podem ser executadas pelo objeto.

Fazendo um modelo próximo do cotidiano, um carro, a classe seria o escopo de um carro, tendo os atributos: Velocidade, cor, tamanho, modelo. E os métodos: Acelerar e freia. Cada carro individualmente seria uma instância dessa classe, um objeto.

2.5.2 Programação Orientada a Aspectos

Programação Orientada a Aspectos, também conhecida como POA [15], é um paradigma de computação, criado por Gregor Kiczales, que serve de complemento ao paradigma de Programação Orientada a Objeto. A POA se consiste em encapsular e inserir em determinados métodos e classes os fragmentos de códigos que não são diretamente ligados à funcionalidade da determinada classe ou método, porém que são requisitos do sistema, e que, além disso, muitas vezes são repetitivo.

Um exemplo de uso de POA seria no uso de geração de linhas em arquivos de logs ou simplesmente medir o tempo de execução dos métodos desejados. A implementação dessas duas funcionalidades diretamente dentro dos dos métodos correspondente deixaria o código poluído e de difícil manutenção, de maneira que o método executasse mais coisas que o funcionamento para qual o mesmo foi designado. Além disso, outro fator que dificultaria a manutenção seria ter que alterar esse mesmo código em muitos locais diferentes, tendo em vista que se deseja gerar logs e medir o tempo de execução em muitos métodos. Para isso foi criado a POA, os fragmentos de código de gerar logs e medir o tempo de execução seriam mantidos em um único lugar e seriam diretamente inserido nos métodos e classes desejados, dessa forma, mantendo o código mais legível e com maior manutenibilidade.

2.5.3 Ambientes de desenvolvimento

Um padrão de configuração de ambientes de desenvolvimento muito comum e utilizados por muitos frameworks, entre eles o Ruby on Rails [6], é a separação em 4 ambientes: , Development, Testing e Staging, Producton. Na implementação desse trabalho, o ambiente que daremos destaque é o ambiente de desenvolvimento, abordado no próximo capítulo.

Development

Development é um ambiente isolado, exclusivo para o uso de cada desenvolvedor individualmente, rodando em sua própria máquina. Nesse ambiente o desenvolvedor testa manualmente e desenvolve as funcionalidades requeridas.

Testing

Testing é um ambiente montato para rodar, exclusivamente, os testes automatizados construídos. Cada teste funciona de modo isolado, para isso, o banco de dados do servidor de Testing é recriado a cada novo teste.

Staging

Staging é o ambiente mais próximo possível do ambiente de produção. Após o termino do desenvolvimento e de realizar os testes de uma funcionalidade, a mesma é colocada em um servidor em nuvem, onde é realizado testes finais por uma equipe de homologação, antes de ser oficialmente lançado.

Production

Production é o ambiente final, em que é disponibilizado para todos os usuário reais. Esse ambiente deve ser o mais estável possível, de maneira que possua a menor quantidade de bugs possíveis.

Capítulo 3

Visão Geral da Proposta de Solução

Nesse capítulo apresentamos uma visão geral sobre a arquitetura e o funcionamento da biblioteca, sem adentrar a fundo nos detalhes da implementação e tecnologias utilizadas, sendo essas partes abordadas nos Capítulos 4 e 5.

Nosso objetivo é construir uma biblioteca que monitore o uso manual de um sistema, e a partir disso, gere teste de regressão de forma automatizada que simulam de forma isolada a ação nas mesmas condições que foram executadas. Na Figura 3.1 podemos ver como são estruturados os 3 módulos da biblioteca: Módulo de monitoramento, Módulo de geração de logs e o Módulo de parser sintático dos logs.

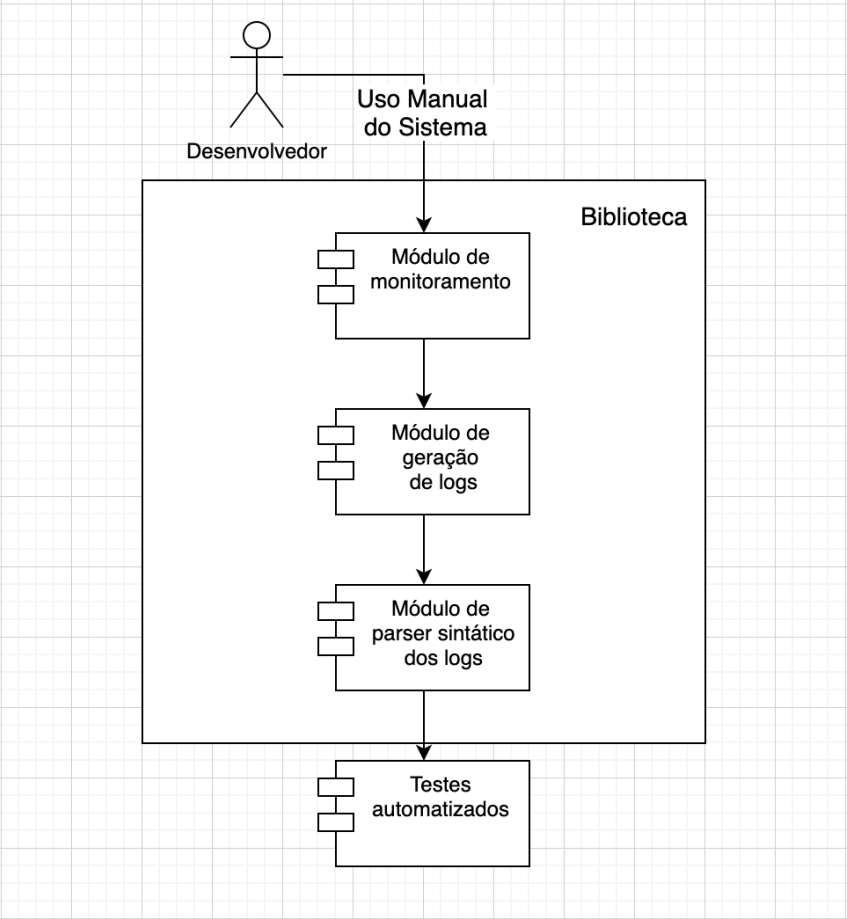

Figura 3.1: Arquitetura da Biblioteca.

3.1 Módulos

Como premissa para o correto funcionamento da biblioteca, apesar de ser voltada para a geração de testes, ela não é executada no ambiente de testes nem no ambiente de produção, e sim no ambiente de desenvolvimento, gerando os testes, que por sua vez, serão executados no ambiente de testes. Essa decisão de ser executada em ambiente de desenvolvimento e não em ambiente de produção foi tomada para se evitar alguns problemas, como por exemplo:

- O problema de concorrência, de muitos usuários utilizando o sistema ao mesmo tempo, fazendo com que os logs gerados ficassem misturados.
- O número excessivo de testes. Se a biblioteca for rodada em um sistema em produção, e cada ação do usuário for convertida em um cenário de teste, o número de teste se tornaria muito alto, repetitivo e de lenta execução.

- A terceira razão seria o sigilo, pois muitos módulos de sistemas lidam com informações que não devem ser públicas, e caso a biblioteca registrasse o passo a passo executado pelos usuários, assim como os valores inseridos pelos mesmos, poderia gerar grandes problemas.
- O grande número de linhas existentes em uma tabela de um banco de dados em produção, em muitos casos passando da casa dos milhares. Esse grande número de elementos, deixaria a execução individual de cada teste muito lenta.

Para evitar esses problemas a biblioteca foi projetada para rodar em ambiente de desenvolvimento, em um servidor localhost na máquina de cada desenvolvedor, possuindo um próprio banco de dados com informações não relevantes.

3.1.1 Módulo de monitoramento

Apesar de todas as etapas serem importantes para o resultado final, a etapa de monitoramento é o coração da biblioteca. Essa etapa do processo é responsável por monitorar cada requisição ao backend, obtendo dessa maneira todos os parâmetros e variáveis necessários para a construção dos testes.

Mas quais parâmetros são esses? Analisando a estrutura de um teste de regressão que foca no comportamento vemos que ele funciona como um teste de caixa preta. Em outras palavras, todo interior e a modularização do código é abstraída, levando apenas em conta a ação executada e o resultado esperado. Por exemplo, em um sistema que houvesse um *endpoint* hipotético que retornasse um usuário no formato JSON [10] a partir de um id, poderíamos ter uma estrutura parecida com essa:

Requisição

```
$ curl -X GET http://localhost:3000/users/2
```

Resposta

```
{
    id: 2,
    name: "Rodrigo Teste",
    username: "teste123",
    email: "teste@gmail.com"
}
```

Olhando para o exemplo acima como as únicas informações que temos sobre o endpoint, podemos concluir que apesar de não sabermos: os detalhes da implementação, quantas classes foram acessadas, quantos arquivos foram acessados e a responsabilidades

de cada um desses componentes, sabemos que foi realizada uma requisição do tipo *get* para "http://localhost:3000/users/2"e que em algum momento foram realizadas uma ou mais consultas no banco de dados e, sabemos também, a resposta que é esperada nessas condições.

Dessa forma, assumindo o diagrama de uma arquitetura de teste descrita na Figura Figura 3.2, verificamos que os parâmetros que o módulo de monitoramento deve identificar para compor os cenários de teste são:

- Todos os elementos presentes no banco de dados no momento que foi realizada a requisição. Por não sabermos necessariamente quais *queries* são executadas no interior do código, a falta de algum elemento pode afetar o resultado, mesmo que o elemento faltante não esteja diretamente ligado com a resposta retornada pelo *endpoint*. Para evitar esse problema, em cada cenário de teste é realizada uma cópia do banco de dados no momento da requisição.
- O método utilizado na requisição: GET, POST, DELETE, entre outros.
- A URL utilizada na requisição.
- Os parâmetros enviados na requisição, contendo por exemplo, um *token* de autenticação, campos de um formulário, filtros de uma busca, entre outros.
- Por fim, a resposta esperada que será enviada na requisição.

Para tal fim, por meio de programação orientada a aspectos, esse monitoramento se da por injeção de código de aspectos nos componentes responsáveis por receber e responder as requisições. Esse código injetado será o responsável por identificar e armazenar as variáveis descritas. Em um sistema orientado a objetos, que segue o padrão MVC [16], como é o caso do framework Ruby on Rails [6], que será abordado no próximo Capítulo 4, os responsáveis por receber e responder as requisições são os *controllers*.

Dessa forma bastaria injetar em todos os *controllers*, ou no *aplicationController* (que é o *controller* que todos as outras classes de *controllers* herdam) um código que ao ser executado, mapeasse e armazenasse todos os parâmetros citados para a construção de um teste. Para outras linguagens e frameworks, a ideia seria análoga, sendo necessário identificar tal componente responsável pelas requisições e por meio de aspectos, inserir um trecho de código.

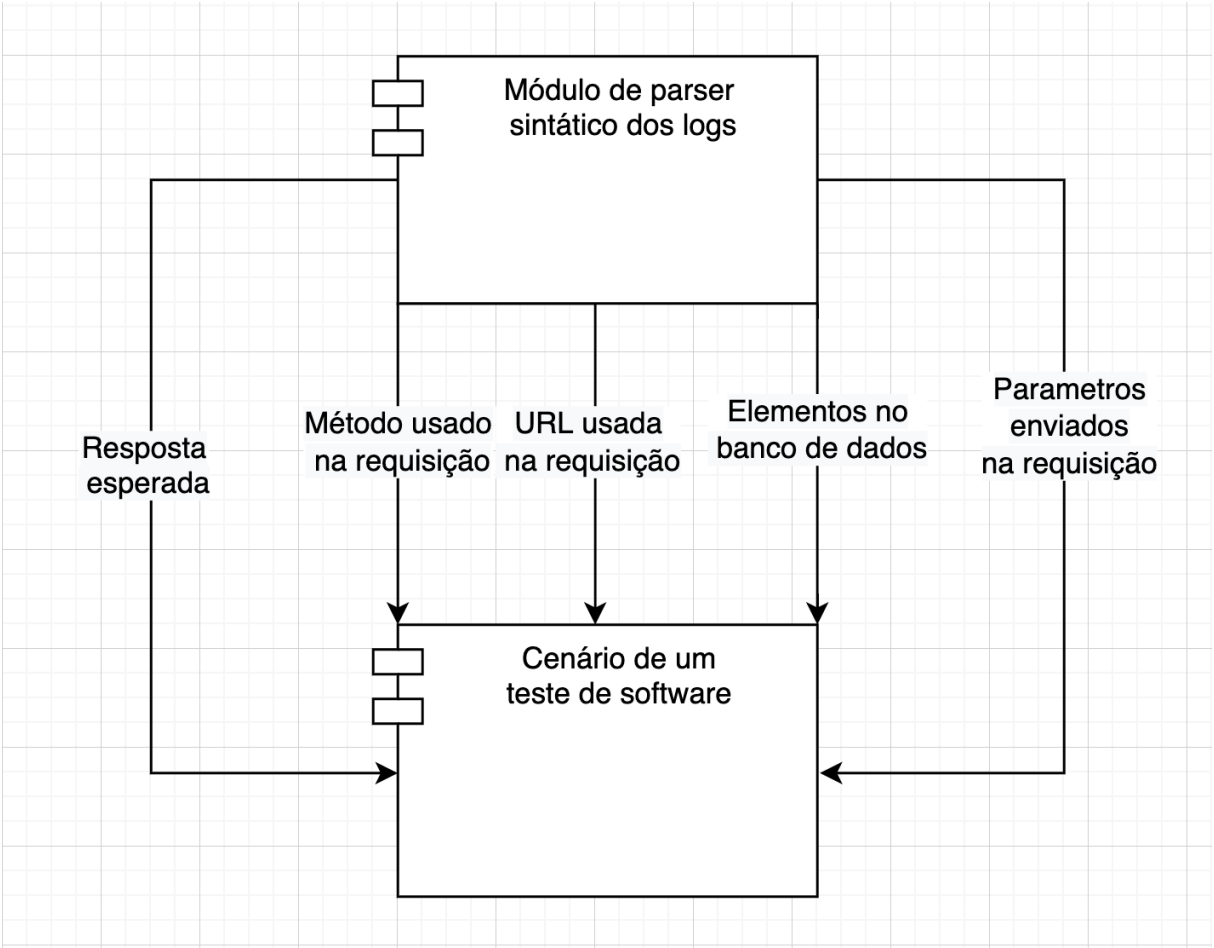

Figura 3.2: Arquitetura de um teste na biblioteca descrita.

3.1.2 Módulo de geração de logs

Após o módulo de monitoramento identificar e armazenar em variáveis os parâmetros citados acima, esses parâmetros são enviados para o módulo de geração de logs, que tem como finalidade persistir e organizar essas informações.

Para realizar tal ação o módulo de geração de logs cria um arquivo de logs dentro da pasta do projeto, ou caso exista, abre ele no modo de edição. Cada vez que o módulo for acessado ele adiciona no final do arquivo os dados recebidos, de acordo com uma formatação específica. Desde que um padrão seja seguido, a formatação de como esse log deve ser escrita é bem flexível. Um exemplo de um log poderia ser:

```
1 novo_log
2 metodo: GET
3 url: http://localhost:3000/users/2
4 parametros: {}
5 elementos_no_banco
```

```
6 user {id: 2, name: "Rodrigo Teste", username: "teste123", email: "
    teste@gmail.com"}
7 fim_elementos_no_banco
8 resposta: {id: 2, name: "Rodrigo Teste", username: "teste123", email: "
    teste@gmail.com"}
9 fim_novo_log
```

Apesar desse exemplo ser um pouco mais simples do que o log utilizado na codificação da biblioteca, que será vista no Capítulo 5, ele satisfaria as necessidades de armazenar as informações vindas do módulo de monitoramento. Sendo assim utilizados no próximo módulo.

3.1.3 Módulo de parser sintático dos logs

Agora que temos o nosso arquivo de logs construido, resta transformar esses logs em testes automatizados. Para isso, quando for desejado pelo desenvolvedor, ele pode executar o Módulo de parser sintático dos logs.

Esse módulo possui duas funções, fazer um parser das linhas existentes no log para recuperar as informações, e após isso, gerar os cenários de testes com elas.

Para realizar o parser sintático é aberto o arquivo dos logs em modo leitura, e lida uma linha de cada vez no formato de string. As linhas entre a linha "novo_log"e "fim_novo_log"indicam um cenário de teste. As linhas numeradas entre 2 e 4 significam respectivamente, o método, a url e os parâmetros enviados na requisição. Entre as linhas "elementos_no_banco"e "fim_elementos_no_banco"são representados os elementos no banco junto de suas características. Por fim, a linha representada pelo número 8 é a resposta da requisição.

Após recuperar todos os valores, eles devem ser inseridos em um arquivo com a sintaxe de alguma linguagem de teste, como por exemplo, na linguagem de Gherkin [17], reconhecida pelo framework de teste Cucumber [18], que é um framework que possui suporte em diversas linguagens. Abordaremos mais a fundo seu funcionamento nos próximos capítulos.

Capítulo 4

Acervo de Bibliotecas

Nesse capítulo falaremos sobre as diferentes bibliotecas e frameworks que foram utilizados no desenvolvimento da biblioteca proposta, descrita no capítulo.

4.1 Ruby e Gemas

Ruby [13] é uma linguagem *open source* desenvolvida em 1995 por Yukihiro "Matz" Matsumoto. A linguagem se popularizou em 2006, e hoje está entre as 20 linguagens no mundo que mais crescem, principalmente devido ao framework Ruby on Rails.

Ruby é uma linguagem interpretada e orientada a objeto, sendo este um dos seus grandes diferenciais pois tudo é um objeto. Um número inteiro por exemplo, é um objeto da classe "Integer". Uma string é um objeto e todos os outros primitivos, também são objetos, ambos com seus próprios métodos e atributos.

Como a maioria das linguagens de programação, Ruby possui um grande acervo de bibliotecas, chamadas de gemas [5]. Para instalá-las foi criado o RubyGems [19] que é o gerenciador de pacotes de sistemas Ruby. Por meio desse gerenciador pode-se instalar qualquer gema pela linha de comando "gem install [gem]". Na implementação da biblioteca descrita, o Ruby será a linguagem utilizada, portando a biblioteca será uma gema Ruby.

4.2 Ruby on Rails

Ruby on Rails [6] é um framework da linguagem Ruby criado para o desenvolvimento web. O framework possui uma própria arquitetura e estrutura baseadas nas filosofias "Don't Repeat Your Self", que significa manter sempre que possível um código centralizado em um lugar só, sem ambiguidades, e "Convention Over Configuration"que sugere o uso das convenções padrões do Rails em vez de configurar à sua maneira, tendo como benefício

principal maior agilidade de desenvolvimento e facilidade ao dar manutenção no código de outro desenvolvedor.

O Ruby on Rails segue uma arquitetura MVC [16], *Model*, *View*, *Controller*. O *model* é onde reside a lógica principal da aplicação, ligação com o banco de dados, validações de atributos entre outros. O *Controller* é o responsável por lidar com as requisições, chamando os métodos construídos no *model*, e por fim, respondendo com alguma *View*. A *View* é a resposta da requisição, que é formatada para envio ao usuário.

Por padrão o Ruby on Rails funciona como *Fullstack*, lidando com o *Frontend* e o *Backend*, retornando para o usuário final um arquivo HTML. Porém, com o crescimento da popularidade de frameworks de *Frontend* como o React.js [20], o Ruby on Rails API começou a ganhar maior destaque. Esse uso como API permite que o framework seja usado exclusivamente como *backend*, retornando para o cliente não um arquivo html, mas sim um arquivo de dados JSON [10], deixando a cargo do Frontend formatar como serão exibidos esses dados. Neste trabalho o Ruby on Rails foi utilizado como o framework base para o uso da gema implementada.

4.3 Cucumber-Rails

Cucumber-Rails [21] é uma gema Ruby que permite utilizar a ferramenta Cucumber [18] junto do framework Ruby on Rails. A ferramenta Cucumber permite que sejam executados cenários de teste de maneira mais simples por meio da linguagem Gherkin [17]. Essa linguagem se baseia em algumas palavras chaves para definir os cenários de testes, sendo elas: "Feature", "Scenario", "When", "Given", "And"e "Then". Um exemplo de cenários de teste escrito com o Cucumber poderia ser da seguinte forma:

```
1  Feature: Guess the word
2
3    # The first example has two steps
4    Scenario: Maker starts a game
5      When the Maker starts a game
6      Then the Maker waits for a Breaker to join
7
8    # The second example has three steps
9    Scenario: Breaker joins a game
10     Given the Maker has started a game with the word "silky"
11     When the Breaker joins the Maker's game
12     Then the Breaker must guess a word with 5 characters
```

A biblioteca descrita no Capítulo 3 se consiste em gerar testes automatizados a partir do uso manual da plataforma, porém ela não se encarrega da execução desses teste. Para

suprir esse necessidade é utilizado algum framework de testes já existente, sendo no caso dessa implementação em particular, foi escolhido o framework Cucumber [18].

4.4 Rspec

Rspec [22] é uma gema para realizar testes na linguagem Ruby. A gema geralmente é utilizada na construção de testes unitários, porém é bastante utilizada também na construção de testes de comportamento. Um exemplo do uso do Rspec seria:

```
RSpec.describe Item, :type => :model do
    let(:item) { create(:item, value: 20, name: "item1") }

    it "is valid with valid attributes" do
        expect(item).to be_valid
    end

    it "should have value equal 20" do
        expect(item.value).to be(20)
    end
end
```

No exemplo acima, por meio da instrução *let* é criada uma variável chamada item, que é uma instância do *model* Item, possuindo atributos *value* igual a 20 e *name* como "item1". Os testes são definidos pela instrução *it*, onde no primeiro caso de teste é testado se o objeto foi criado, e no segundo caso de teste, é testado se o atributo *value* possui valor 20. Pela seu grande arcabouços de ferramentas de testes, o Rspec auxilia o Cucumber na criação dos testes automatizados.

4.5 Aquarium

Aquarium [23] é uma gema que permite utilizar Aspect-Oriented Programming (AOP) dentro do Ruby e do framework Ruby on Rails. Um exemplo de código de utilização da gema seguindo a própria documentação seria:

```
require 'aquarium'
Aspect.new :around, :calls_to => :all_methods, :on_types => [Foo, Bar] do |
    join_point, object, *args|
  p "Entering: #{join_point.target_type.name}##{join_point.method_name} for
     object #{object}"
  result = join_point.proceed
  p "Leaving: #{join_point.target_type.name}##{join_point.method_name} for
    object #{object}"
```

```
  result  # block needs to return the result of the "proceed"!
end
```

No dado exemplo, assumindo as classes "Foo"e "Bar", quando qualquer método dessas classes fosse chamado, antes do método ser executado iria ser impresso a string "Entering...". Após, ao executar "join_point.proceed"o método seria executado. Por fim, após a execução do método, iria ser impresso a string "Leaving...". O Aquarium é uma das gemas chaves para esse projeto, ele que possibilita o uso de aspectos dentro do Ruby on Rails, tornando possível o monitoramento do sistema.

4.6 Database Cleaner

Database Cleaner [24] é uma gema que possibilita limpar o banco de dados. A sua principal utilização é dentro do ambiente de testes, garantindo que o banco de dados de teste esteja sempre vazio entre um teste e outro, garantindo que um teste não influencie no funcionamento do outro, mantendo-os isolados. Neste projeto ele é utilizado jundo do Cucumber para garantir que cada cenário de teste seja independente entre si. Para utilizar a gema, pasta importa-la dentro do ambiente de testes:

```
group :test do
  gem 'database_cleaner-active_record'
end
```

4.7 SimpleCov

SimpleCov [25] é uma biblioteca Ruby [13] que possibilita analisar a cobertura de testes realizados tanto pela ferramenta Cucumber [18] quanto pela ferramenta Rspec [22]. Nesse trabalho nós a utilizaremos para testar a cobertura de teste alcançada pela biblioteca desenvolvida. Ela analisa cobertura em nível de caminho, verificando todas as linhas do código que foram acessadas pelos testes em relação ao total de linhas, gerando um arquivo HTML a taxa de cobertura para cada arquivo de código como mostrado na Figura 4.1. Clicando em um arquivo são apresentados de forma detalhada, utilizando a cor vermelha, os trechos de códigos que não foram testados, conforme a Figura 4.2.

Para instalar a gema SimpleCov [25] em um projeto rails, é necessário adicionar a gema no Gemfile:

```
gem 'simplecov', require: false, group: :test
```

No arquivo de configuração do ambiente de testes é necessário adicionar o seguinte trecho de código:

```
require 'simplecov'
SimpleCov.start 'rails'
```

Por fim, é possível encontrar o arquivo HTML com os detalhes da cobertura no caminho "coverage/index.html"a partir da pasta do projeto Rails [6].

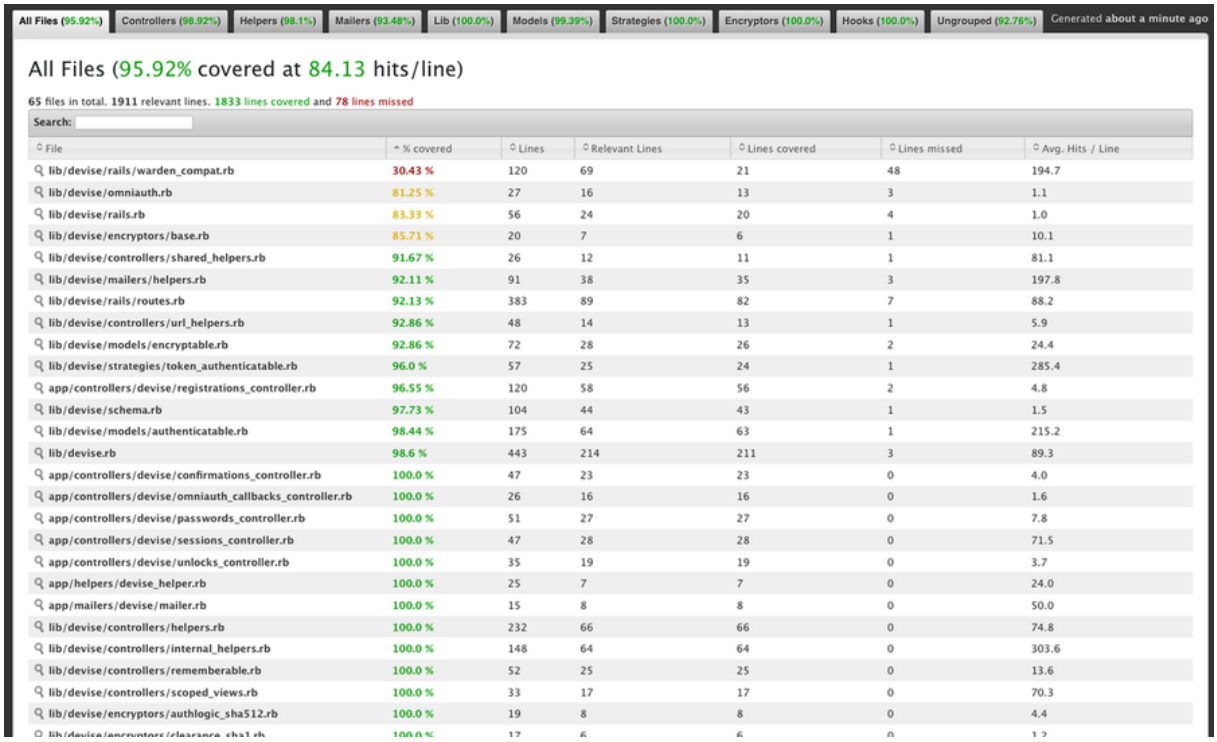

Figura 4.1: Exemplo retirado do site https://github.com/simplecov-ruby/simplecov.

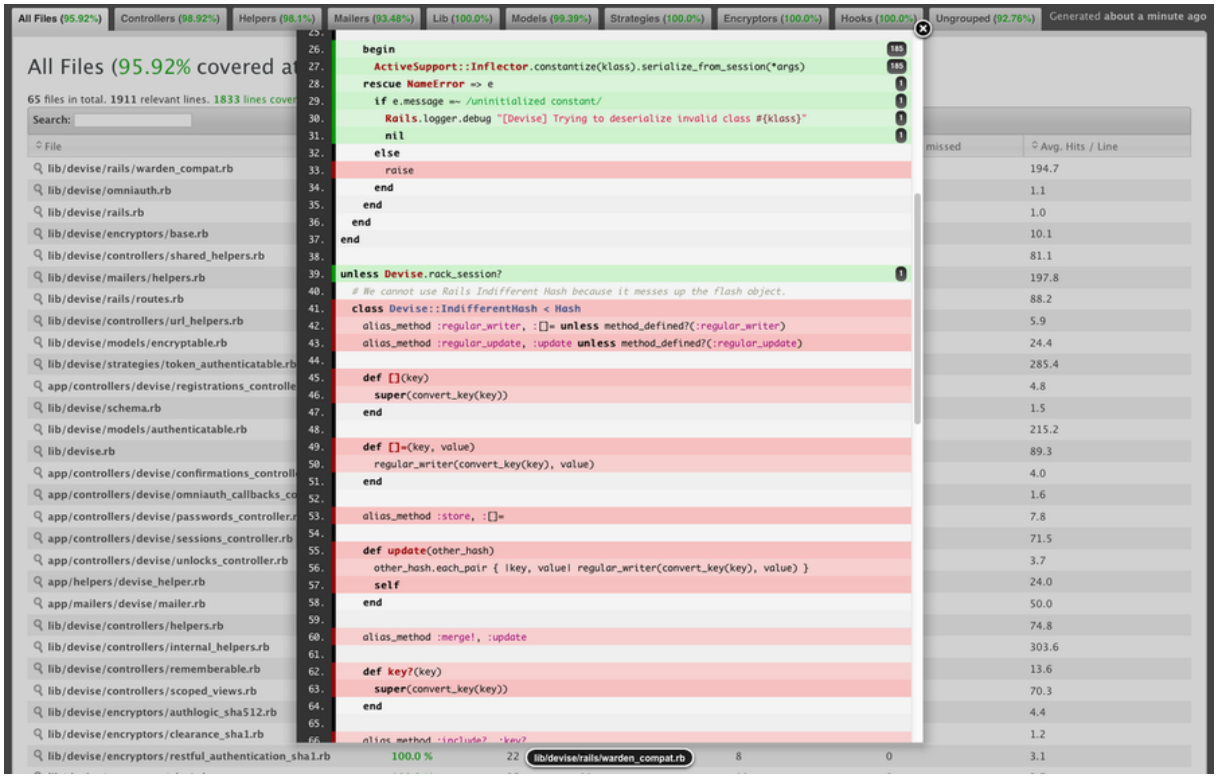

Figura 4.2: Exemplo retirado do site https://github.com/simplecov-ruby/simplecov.

Capítulo 5

Implementação do projeto

Neste capítulo será apresentada a implementação da biblioteca descrita no Capítulo 3, explicando de forma detalhada o código desenvolvido. A abordagem envolveu construir uma gema [5] em Ruby [13] feita para ser utilizada em um ambiente Ruby on Rails API [6], capaz de monitorar o uso do sistema e gerar automaticamente scripts de testes de regressão automatizados, que por fim serão executados pelo framework de testes Cucumber [18]. A gema referida será denominada como Regression Generator, e o código fonte está disponível na URL https://github.com/fredericodib/bdd_generator.

5.1 Implementação

Conforme apresentado no Capítulo 3, a implementação foi realizada em 3 módulos: Módulo de monitoramento, Módulo de geração de logs e Módulo de parser sintático dos logs. Abordaremos de forma detalhada cada um deles.

5.1.1 Módulo de monitoramento

Este módulo é o principal responsável por monitorar as requisições feitas ao backend, separando as principais variáveis que serão utilizadas na montagem dos testes automatizados. O framework Ruby on Rails [6], está estruturado na arquitetura MVC (model, View e Controller) e sabemos que os Controller são os responsáveis por definir o funcionamento de cada requisição feita, de maneira que toda url de API é ligada diretamente a um método de um Controller.

A primeira parte do módulo se baseia em conseguir uma lista com todos os Controllers existentes na aplicação e por meio da gema Aquarium [23], inserirmos código em cada método que eles possuem, utilizando um paradigma orientado à Aspectos. Isso pode ser visto no seguinte bloco de código:

```
1  # templates/aspects.rb
2
3  include Aquarium::Aspects
4
5  controllers = ApplicationController.subclasses
6
7  ...
8
9  Aspect.new(:around,
10             calls_to: :all_methods,
11             for_types: controllers,
12             method_options: :exclude_ancestor_methods) do |jp, obj, *_args
       |
13
14     <CODIGO_QUE_SERA_INSERIDO>
15
16 end
```

Na linha 5 criamos uma variável "controllers"que possui um array com todas as classes de Controllers utilizadas na aplicação. A partir da linha 9 instanciamos um objeto do tipo "Aspect"provido pela biblioteca Aquarium [23], e nele passamos como argumento a variável "controllers"fazendo com que toda vez que algum método de algum Controller seja executado, será executado junto o código:

```
1  # templates/aspects.rb
2
3  ...
4
5  models = ApplicationRecord.subclasses
6
7  models.each do |m|
8      m.all.each do |r|
9          data << "#{m} #{r.id} #{r.to_json}\n"
10     end
11 end
12
13 request_controller = obj.params[:controller]
14 request_action = obj.params[:action]
15 body = obj.params
16 headers = obj.request.headers['Authorization'] ? {Authorization: obj.
       request.headers['Authorization']} : {}
17 fullpath = obj.request.fullpath
18 method = obj.request.method
19 result = jp.proceed
20 status = obj.status
```

```
21
22 ...
```

No trecho de código acima são coletadas todas as variáveis importantes para a geração dos testes. Entre as linhas 7 e 11 é realizado um loop passando por todos os elementos no banco de dados no momento em que o método foi acessado, gerando um vetor chamado "data"contendo esses registros em forma de strings. Nas linhas posteriores são coletados respectivamente o nome do Controller acessado, o tipo de requisição feita, os parâmetros enviados no corpo da requisição, o header da requisição, a URL acessada, o método acessado pelo Controller, a resposta enviada pelo controller e por fim o número do status da resposta.

Como em alguns cenários o desenvolvedor pode não querer que seja feito o monitoramento durante alguma parte do desenvolvimento, para não gerar logs indesejáveis, para dar essa liberdade foi implementada uma *flag* localizada em "config/initializer/aspects.rb"dentro do projeto Rails após a instalação da gema. Ao mudar o valor da *flag* "regression_generator_monitoring_controllers"para "false"o monitoramento será interrompido, e para reativa-lo bastaria voltar o valor para "true", conforme mostrado no fragmento de código abaixo:

```
1 ...
2 regression_generator_monitoring_controllers = true
3 ...
```

5.1.2 Módulo de geração de logs

Neste módulo usamos todas as informações obtidas no Módulo de Monitoramento e geramos um arquivo de logs personalizado com ela. Esses logs são gerados com o seguinte bloco de código:

```
1 # templates/aspects.rb
2
3 ...
4
5 File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "\n\
    nnew_test:\n" }
6 File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    controller: #{request_controller}\n" }
7 File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    action: #{request_action}\n" }
8 File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    body: #{body.to_json}\n" }
9 File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    headers: #{headers.to_json}\n" }
```

```
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    path: #{fullpath}\n" }
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    method: #{method}\n" }
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    data:\n" }
data.uniq.each { |d| File.open('features/regression_generator_logs.txt', 'a
    ') { |f| f.write d } }
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    status: #{status}\n" }
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "
    retorno:\n" }
File.open('features/regression_generator_logs.txt', 'a') { |f| f.write "#{
    result}\n" }

...
```

Cada vez que o bloco de código é executado é adicionado no final do arquivo "regression_ generator_logs.txt"um novo registro de logs com as informações que serão usadas na construção de um cenário de teste de regressão. O inicio de cada registro é demarcado com a linha "new_teste:"significando o inicio de um novo registro e o termino de um antigo. Os próximos 6 elementos são respectivamente: O controller acessado, o método de dentro do controller que foi acessado, o corpo e o cabeçalho da requisição, a URL acessada e o método da requisição. Posteriormente é seguido com o texto "data:", e os elementos que haviam no banco de dados no momento que o controller foi acessado. E por fim, as ultimas três linhas do registro são os status da resposta enviado pelo controller e os dados do retorno enviados.

Exemplo de um arquivo de logs

Em um sistema backend que possui um CRUD [26] de "Itens"possuindo dois campos, nome e valor. Ao fazer 3 requisições, a primeira cadastrando um item com nome e valor respectivamente "item1"e "30". A segunda cadastrando um item com os campos "item2"e "20". E a terceira requisição recuperando todos os itens cadastrados. Seria gerado 3 registros de log:

```
new_test:
controller: items
action: create
body: {"item":{"name":"item1","value":"30"}, "controller":"items
    ", "action":"create"}
```

```
headers: {}
path: /items
method: POST
data:
status: 201
retorno:
{"id":20,"name":"item1","value":30, "created_at":"2021-04-19T02
    :05:18.017Z", "updated_at":"2021-04-19T02:05:18.017Z"}


new_test:
controller: items
action: create
body: {"item":{"name":"item2","value":"20"}, "controller":"items
    ","action":"create"}
headers: {}
path: /items
method: POST
data:
Item 20 {"id":20,"name":"item1","value":30, "created_at
    ":"2021-04-19T02:05:18.017Z", "updated_at":"2021-04-19T02
    :05:18.017Z"}
status: 201
retorno:
{"id":21,"name":"item2","value":20, "created_at":"2021-04-19T02
    :05:27.928Z", "updated_at":"2021-04-19T02:05:27.928Z"}


new_test:
controller: items
action: index
body: {"controller":"items","action":"index"}
headers: {}
path: /items
method: GET
data:
```

```
Item 20 {"id":20,"name":"item1","value":30, "created_at
    ":"2021-04-19T02:05:18.017Z", "updated_at":"2021-04-19T02
    :05:18.017Z"}
Item 21 {"id":21,"name":"item2","value":20, "created_at
    ":"2021-04-19T02:05:27.928Z", "updated_at":"2021-04-19T02
    :05:27.928Z"}
status: 200
retorno:
[{"id":20,"name":"item1","value":30,"created_at":"2021-04-19T02
    :05:18.017Z","updated_at":"2021-04-19T02:05:18.017Z"},{"id
    ":21,"name":"item2","value":20,"created_at":"2021-04-19T02
    :05:27.928Z","updated_at":"2021-04-19T02:05:27.928Z"}]
```

5.1.3 Módulo de parser sintático dos logs

O módulo do parser sintático é o responsável de ler o arquivo de logs criados pelo módulo anterior e gerar testes de regressão na linguagem Gherkin [17] para que sejam executados pelo framework de testes Cucumber [18]. Essa analise do arquivo de logs ocorre quando o desenvolvedor usa o comando "rails regression_generator:build"no terminal na pasta root do projeto. Esse comando é uma tesk que nossa gema insere no projeto durante a instalação, permitindo que os testes só sejam criados quando for de interesse dos desenvolvedores do projeto. O seguinte código é o responsável por executar essa tesk:

```
1 namespace :regression_generator do
2   task build: :environment do
3     RegressionGeneratorParser.build
4   end
5 end
```

Quando a task é executada, ela chama o método "build"da classe "RegressionGeneratorParser". Essa classe é a responsável ler linha por linha do arquivo de logs e gerar o arquivo de testes em "features/regression_generator.feature", possuindo 5 tipos principais de comandos Gherkin:

- O primeiro tipo é responsável por delimitar o cenário de teste, possuindo o escopo "Scenario: <controle acessado> <método do controller>"
- O segundo tipo tem a finalidade de listar os elementos junto de seus atributos existente no banco de dados registrados pelo log. O comando possui o escopo "Given There is an instance of <Nome do elemento> with id <Id do elemento> and params: <Atributos do elemento>".

- O terceiro tipo é responsável por especificar qual a ação executada, que no caso seria o acesso a uma API do sistema. Portanto essa instrução deve conter a URL e o método usados e os parâmetros enviados no corpo e no cabeçalho da requisição. O comando possui o escopo "When the client requests with <método da requisição> <URL>, body: <body>, headers: <header>".
- O quarto tipo é responsável por delimitar o status da resposta esperado pelo teste, possuindo o escopo "Then the response status should be <status da resposta>"
- O quinto e último tipo é responsável por delimitar o corpo da resposta esperada pela raquisição no cenário definido. O comando possui o formato "And The JSON response should be <corpo da resposta>".

Mesmo estando com as instruções de testes prontas, elas não definem como deve ser executados os testes, sendo papel da gema também especificar o que o Cucumber deve executar ao ler cada uma das instruções descritas acima. Para isso a gema gera um arquivo chamado "regression_generator_steps.rb"que é responsável por detalhar o funcionamento de cada passo de um cenário de teste. Dentro desse arquivo podemos encontrar os seguintes blocos de código:

```
Given(/^There is an instance of (.*?) with id (.*?) and params: (.*?)$/) do
    |obj, id, params|
  if !eval(obj).find_by_id(id)
    o = eval(obj).new(JSON.parse(params))
    o.save(:validate => false)

  end
end
```

O bloco acima se refere as expressões de segundo tipo, ele instancia por instanciar o objeto descrito na instrução e salvalo no banco de dados. A gema também utiliza outra gema chamada Database Clear [24] responsável por zerar todos os dados armazenados no banco de dados do ambiente de testes, garantindo que todos os testes sejam totalmente isolados.

```
When(/^the client requests with (GET|POST|PUT|DELETE|PATCH) (.*?), body:
   (.*?), headers: (.*?)$/) do |request_type, path, params, headers|
  if JSON.parse(headers)['Authorization']
    header 'Authorization', JSON.parse(headers)['Authorization']
  end

  case request_type
  when 'GET'
    get path, JSON.parse(params)
  when 'POST'
```

```
    post path, JSON.parse(params)
  when 'PUT'
    put path, JSON.parse(params)
  when 'DELETE'
    delete path, JSON.parse(params)
  when 'PATCH'
    patch path, JSON.parse(params)
  end
end
```

Essa instrução se refere ao terceiro tipo, ela recupera do texto escrito na instrução do passo e recria a requisição feita com os mesmos parâmetros da original feita manualmente.

```
Then(/^the response status should be (.*?)$/) do |status|
  expect(last_response.status).to eq status.to_i
  rescue RSpec::Expectations::ExpectationNotMetError => e
  puts 'Response body:'
  puts last_response.body
  raise e
end
```

Esse bloco se refere a quarto tipo de instrução, ele recupera a resposta da requisição gerada no passo anterior e compara o status da resposta esperada com o status realmente obtido, gerando uma falha no teste caso sejam diferentes.

```
And(/^The JSON response should be (.*?)$/) do |value|
  def remove_key(obj, key)
    if obj.respond_to?(:key?) && obj.key?(key)
      obj.delete(key)
    elsif obj.respond_to?(:each)
      obj.each { |a| remove_key(a, key) }
    end
  end
  expected_json = JSON.parse(value)

  true_json = JSON.parse(last_response.body)
  expect(true_json).to eq expected_json
end
```

Por fim o código acima se refere ao quinto e ultimo tipo de instrução, nele é recuperado o corpo da requisição feita na instrução de número 3 e é comparado com o resultado esperado, gerando uma falha de testa caso os resultados não sejam iguais.

Todos esses trechos de código serão utilizados pelo framework Cucumber [18], que por fim executará todos os testes.

Exemplo de um arquivo de testes gerados.

Para demonstrar ná prática como ficariam esses arquivos com os cenários de testes, usaremos os exemplos de logs mostrado na sessão 5.1. Ao executar o comando "rails regression_generator:build"o módulo de parser será ativado, lendo o arquivo de log e gerando a partir dele o arquivo com todos os cenários de testes prontos para serem executados pelo Cucumber. O resultado seria 3 cenários de testes que podem ser vistos abaixo:

```
Scenario: items create
  When the client requests with POST /items, body: {"item":{"name":"item1",
   "value":"30"},"controller":"items","action":"create"}, headers: {}
  Then the response status should be 201
  And The JSON response should be {"id":20,"name":"item1","value":30,"
   created_at":"2021-04-19T02:05:18.017Z","updated_at":"2021-04-19T02
   :05:18.017Z"}

Scenario: items create
  Given There is an instance of Item with id 20 and params: {"id":20,"name":
   "item1","value":30,"created_at":"2021-04-19T02:05:18.017Z","updated_at":
   "2021-04-19T02:05:18.017Z"}
  When the client requests with POST /items, body: {"item":{"name":"item2",
   "value":"20"},"controller":"items","action":"create"}, headers: {}
  Then the response status should be 201
  And The JSON response should be {"id":21,"name":"item2","value":20,"
   created_at":"2021-04-19T02:05:27.928Z","updated_at":"2021-04-19T02
   :05:27.928Z"}

Scenario: items index
  Given There is an instance of Item with id 20 and params: {"id":20,"name":
   "item1","value":30,"created_at":"2021-04-19T02:05:18.017Z","updated_at":
   "2021-04-19T02:05:18.017Z"}
  Given There is an instance of Item with id 21 and params: {"id":21,"name":
   "item2","value":20,"created_at":"2021-04-19T02:05:27.928Z","updated_at":
   "2021-04-19T02:05:27.928Z"}
  When the client requests with GET /items, body: {"controller":"items","
   action":"index"}, headers: {}
  Then the response status should be 200
  And The JSON response should be [{"id":20,"name":"item1","value":30,"
   created_at":"2021-04-19T02:05:18.017Z","updated_at":"2021-04-19T02
   :05:18.017Z"},{"id":21,"name":"item2","value":20,"created_at":"
   2021-04-19T02:05:27.928Z","updated_at":"2021-04-19T02:05:27.928Z"}]
```

- No primeiro cenário não há nenhum elemento pré existente no banco de dados, é feita uma requisição do tipo POST para /items, com os parâmetros de criação do item, nome: "item1", valor: 30. A requisição devolve como resposta os dados do

item criado. por fim o teste compara se a resposta esperada é igual a resposta recebida.

- No segundo cenário já possui um item no banco de dados, sendo ele, o item descrito no teste anterior, Vale ressaltar que os testes são independente entre si, o banco de dados não é persistido entre dois cenários, portanto a existência desse item é definida pela primeira instrução do cenário. Posteriormente é feita uma requisição do tipo POST para /items, com os parâmetros de criação do item, nome: "item2", valor: 20. A requisição devolve como resposta os dados do item criado. por fim o teste compara se a resposta esperada é igual a resposta recebida.
- Por fim no último cenário declara que existe dois itens no banco de dados, ao fazer uma requisição do tipo GET para /items, é retornado uma lista com todos os items existentes, posteriormente o teste compara se a resposta esperada é igual a resposta recebida.

Os testes já estão prontos para serem executados pelo framework Cucumber [18]. Para isso deve ser utilizado o comando "cucumber"na pasta do projeto. Após a execução dos testes será apresentado o número de cenários que falharam e passaram, como apresentado abaixo:

```
... 3 scenarios (3 passed)
12 steps (12 passed)
0m0.108s
```

5.2 Instalação da gema e instruções da gema

A instalação da gema para ser utilizado em um projeto Ruby on Rails [6] é bastante simples. O primeiro passo é adicionar o seguinte bloco de código ao arquivo "gemfile":

```
group :development, :test do
  gem 'cucumber-rails', require: false
  gem 'database_cleaner'
  gem 'rspec'
  gem 'aquarium'
  gem 'regression_generator', '1.0', git: "https://github.com/fredericodib/
    regression_generator", branch: "main"
end
```

Ele é o responsável por importar a gema e todas as outras dependências. Posteriormente é necessário instalar o Cucumber e a gema RegressionGenerator, utilizando os comandos:

```
$ rails generate cucumber:install
$ rails g regression_generator:install
```

Após isso a gema já estará funcionando e gerando os logs de cada requisição. Para gerar os testes a partir do arquivo de logs o desenvolvedor deve utilizar o comando "rails regression_generator:build", por fim para executar os testes deve ser utilizado o comando "cucumber".

Capítulo 6

Casos de Uso

Nesse Capítulo nosso objetivo é testar a gema proposta e desenvolvida, Regression Generator, em aplicações existentes e dessa forma avaliar os resultados que a gema oferece. Para tal fim, usamos três sistemas desenvolvidos em Ruby on Rails API [6], o primeiro sendo um sistema bem simples, de gerenciamento de contatos, o segundo, um blog e o terceiro o sistema de uma startup chamada Sisterwave.

Para entendermos corretamente o potencial da gema é importante entendermos a fundo os sistemas para os quais iremos usa-la, seu banco de dados, regras de negócio, quais testes iremos realizar manualmente e o resultado que obtivemos. Inicialmente ambos os sistemas estarão sem nenhum registro de teste. Por meio do aplicativo Postman [27], que é um aplicativo feito para acessar API's, por meio de uma interface gráfica, iremos realizar nossos testes manuais. Após a realização dos testes utilizaremos a gema Cucumber [18] para rodar os testes e vermos o número de cenários testados, número de passos, e tempo de execução. Posteriormente usaremos a gema SimpleCov [25] para vermos a cobertura de teste alcançada. Por fim, analisaremos o tempo necessário para executar os testes manuais, em relação ao tempo provável que levaria para codificarmos os mesmos testes automatizados gerados.

6.1 Uso no sistema de contatos

6.1.1 Detalhamento do sistema

O primeiro sistema é um sistema gerenciador de contatos. Ele disponibiliza todos os contatos registrados, assim como a possibilidade de adicionar novos, excluir e editar algum contato existente. O código fonte do sistema pode ser encontrado no Github [28] na url https://github.com/fredericodib/tg_contato.

Banco de dados

Na Figura 6.1 podemos ver que o banco de dados do sistema possui uma única tabela, que é a tabela de contatos. Essa tabela possui um "id"que é a chave primária, os campos "created_at"e "updated_at"que significam respectivamente a data de criação da linha e a data de ultima alteração feita no elemento, e por fim os campos de email e nome.

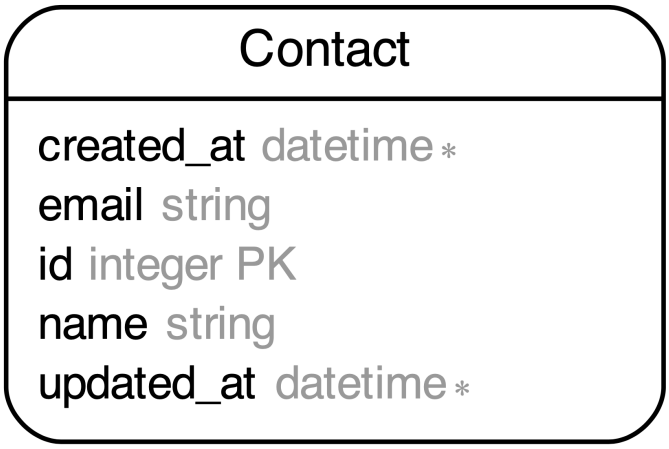

Figura 6.1: Arquitetura do banco de dados do sistema de contatos.

Regras de negócio

Para um registro ser salvo no banco de dados, nem o campo de email e nem o campo de nome podem estar vazios. Além disso, o campo de email precisa ser único e seguir a formatação esperada de um email real. Caso esses requisitos não sejam atendidos o sistema retornará um erro.

API's do sistema

Considerando que o sistema está sendo executado localmente na porta 3000, temos os seguintes endpoints disponíveis para serem acessados no sistema:

```
API para acessar todos os contatos disponíveis:
GET http://localhost:3000/contacts

--------

API para criar um novo contato:
```

```
POST http://localhost:3000/contacts
Body:
[contact]name    exemplo_de_nome
[contact]email   exemplo_de_email

--------

API para editar um contato existente:
PATCH http://localhost:3000/contacts/:id_contato
Body:
[contact]name    exemplo_de_nome
[contact]email   exemplo_de_email

--------

API para acessar um contato em específico:
GET http://localhost:3000/contacts/:id_contato

--------

API para deletar um contato:
DELETE http://localhost:3000/contacts/:id_contato
```

6.1.2 Testes manuais

Através da ferramenta Postman [27] realizamos os seguintes testes manuais:

- Criar alguns contatos com os parâmetros corretos, Figura 6.2.
- Tentar criar um contato enviando parâmetros incorretos, esperando um erro.
- Editar um contato com os parâmetros corretos.
- Tentar editar um contato enviando parâmetros incorretos, esperando um erro.
- Acessar a lista de todos os contatos.
- Acessar um contato específico pelo id.
- Deletar um contato.

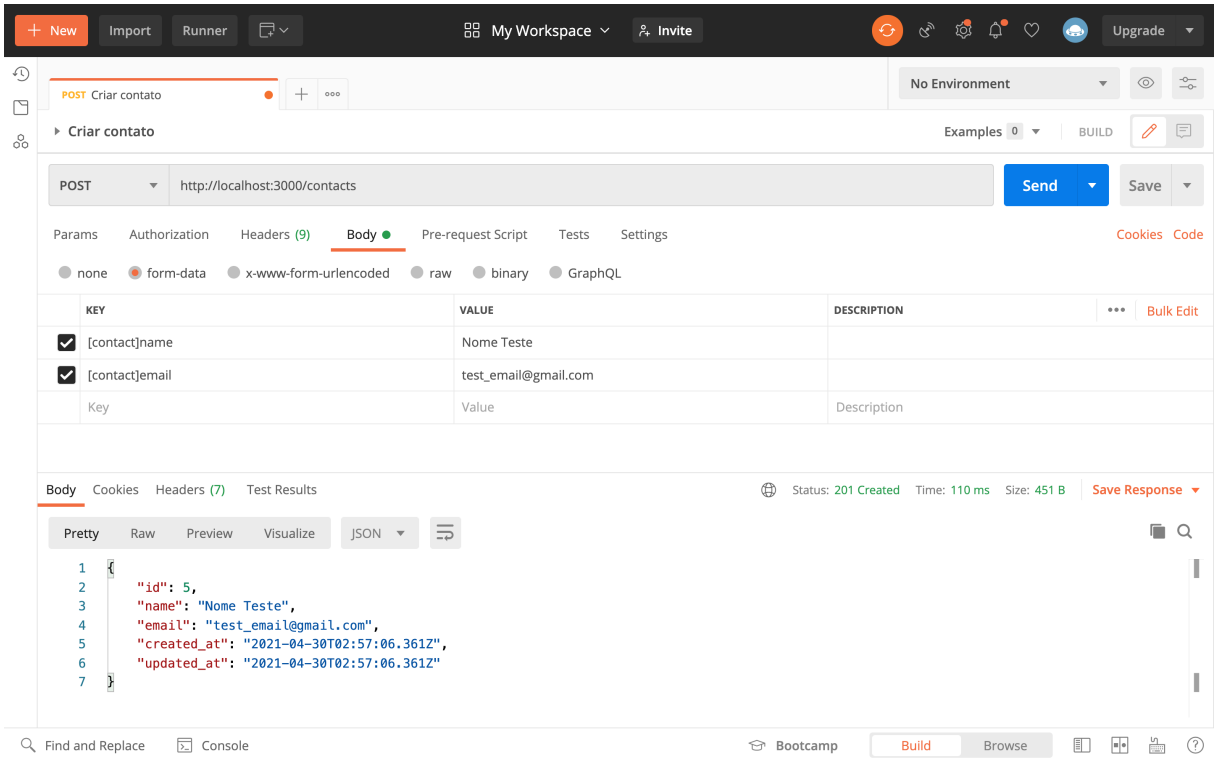

Figura 6.2: Exemplo de um contato criado manualmente através da ferramenta Postman.

6.1.3 Uso da gema Regression Generator

Após realizar os testes manuais listados acima com a gema Regression Generator ativada, foi criado um arquivo de logs conforme mostrado na Figura 6.3 em que mostra um log de um contato criado corretamente e outro de um contato que obteve falha na criação. Ao executar o comando:

```
rails regression_generator:build
```

Todos os testes automatizados são gerados. Pode-se observar um fragmento desses testes na Figura 6.4, em que é apresentado 4 cenários de testes:

- Cenário de falha ao tentar cadastrar um contato sem o email.
- Cenário de falha ao cadastrar um contato com um email que já existe em outro contato.
- Cenário de recuperação de todos os contatos.
- Cenário de recuperação de um usuário específico pelo id.

Por fim, com os testes prontos pode-se usar o comando "bundle exec cucumber"para que a ferramenta Cucumber [18] execute todos os testes automatizados gerados e também

utilizar a ferramenta SimpleCov [25] para obter a cobertura de teste do código. Como apresentado respectivamente nas Figuras 6.5 e 6.6, observa-se que nenhum teste falhou e a cobertura alcançada foi de 100%.

Em resumo, os seguintes resultados foram obtidos no primeiro experimento:

- 100% de cobertura de testes.
- 19 cenários de testes criados.
- 86 passos de testes criados.
- Tempo de execução de todos os cenários de teste: 0.332 segundos.
- Tempo necessário para a execução dos testes manuais pela ferramenta Postman: 2 minutos e 33 segundos.

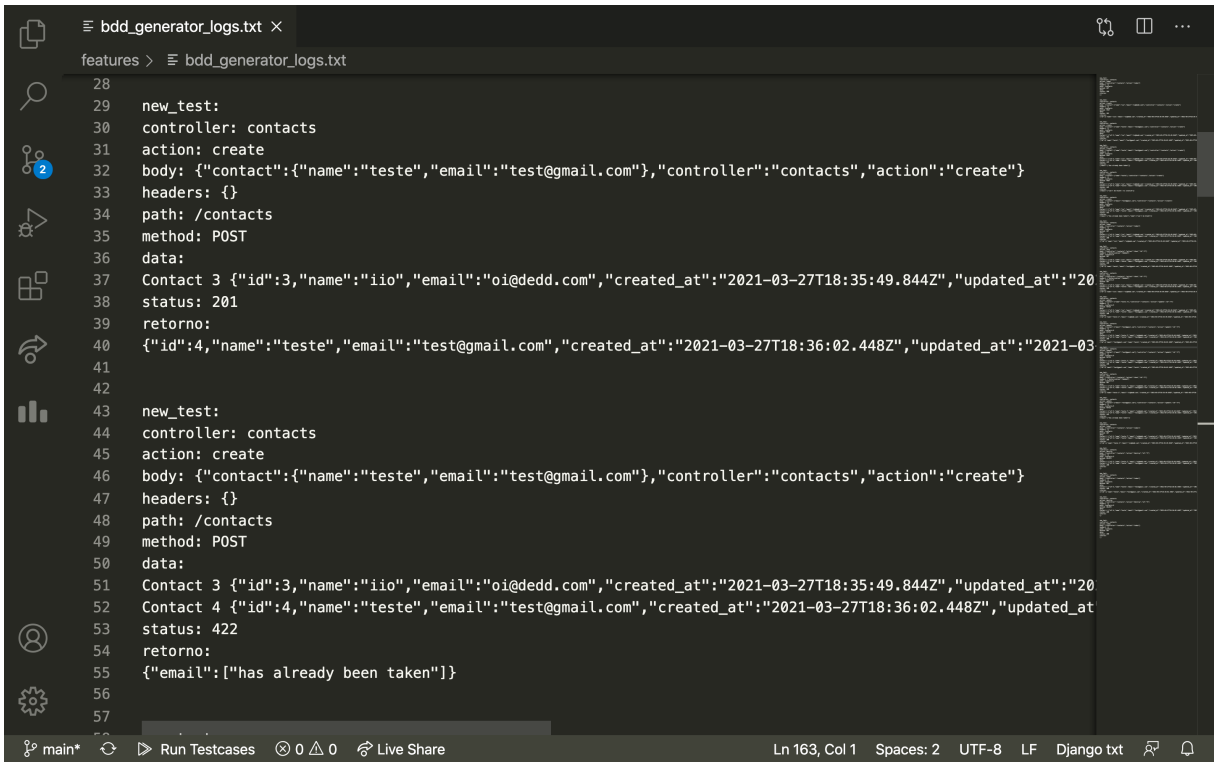

Figura 6.3: Fragmento de logs gerados pela gema Regression Generator no sistema de gerenciamento de contatos.

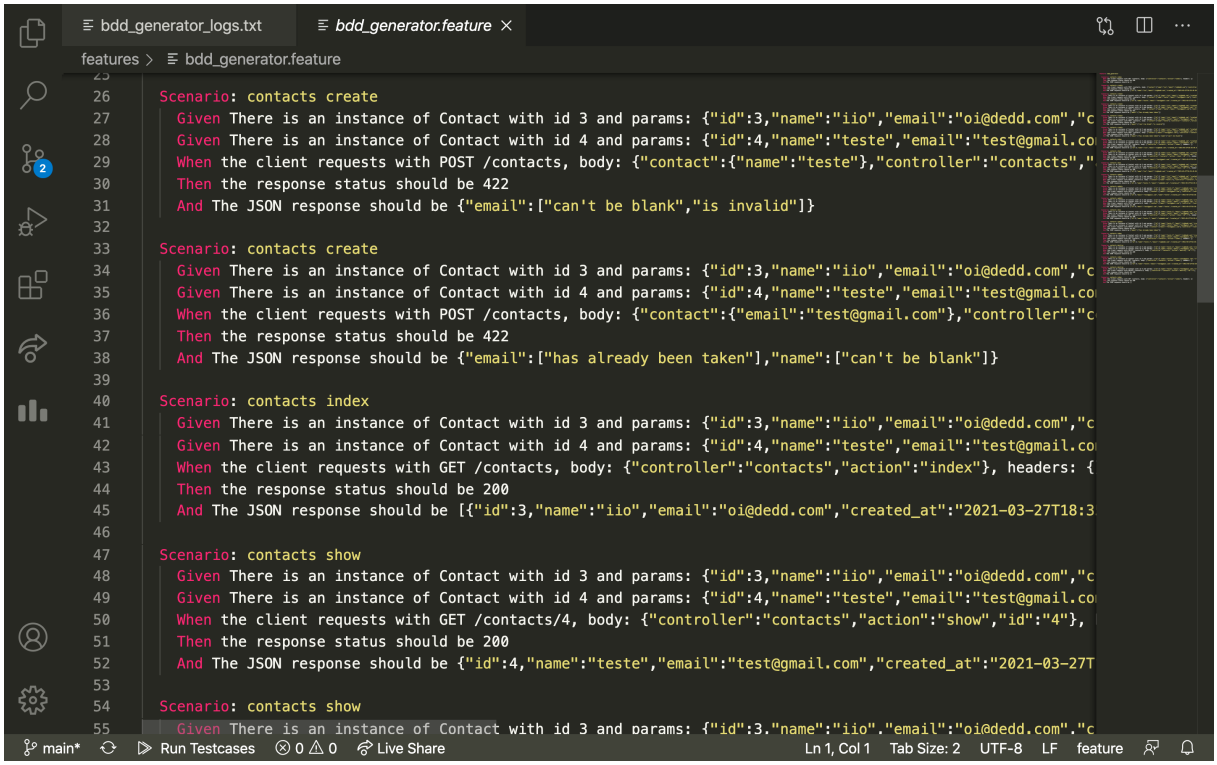

Figura 6.4: Fragmento de testes gerados pela gema Regression Generator no sistema de gerenciamento de contatos.

Figura 6.5: Execução dos testes gerados pela gema Regression Generator no sistema de gerenciamento de contatos.

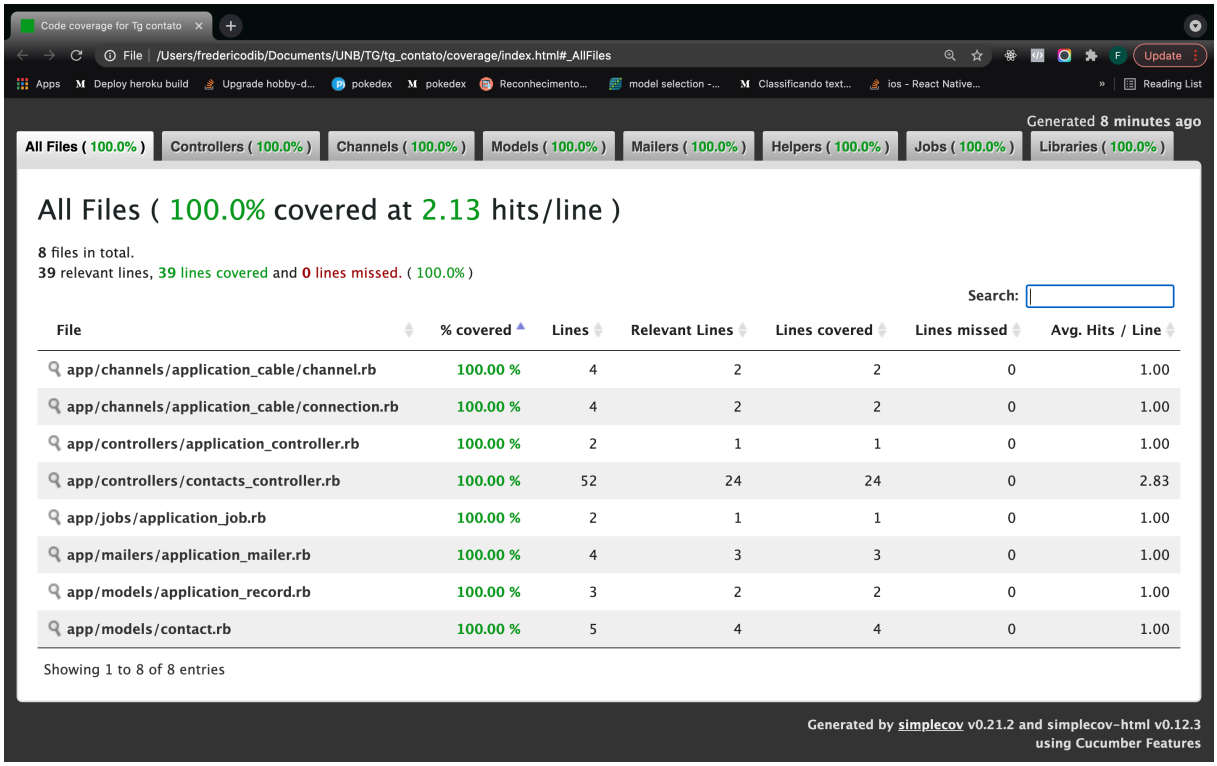

Figura 6.6: Cobertura de testes no sistema de gerenciamento de contatos.

6.2 Uso no sistema de blog

6.2.1 Detalhamento do sistema

O segundo sistema é um sistema de um blog. É um sistema que possui autenticação de usuário, login, registro, criação e edição de postagens, criação de comentários e muito mais. O código fonte do sistema pode ser encontrado no Github [28] na url https://github.com/fredericodib/tg_blog.

Banco de dados

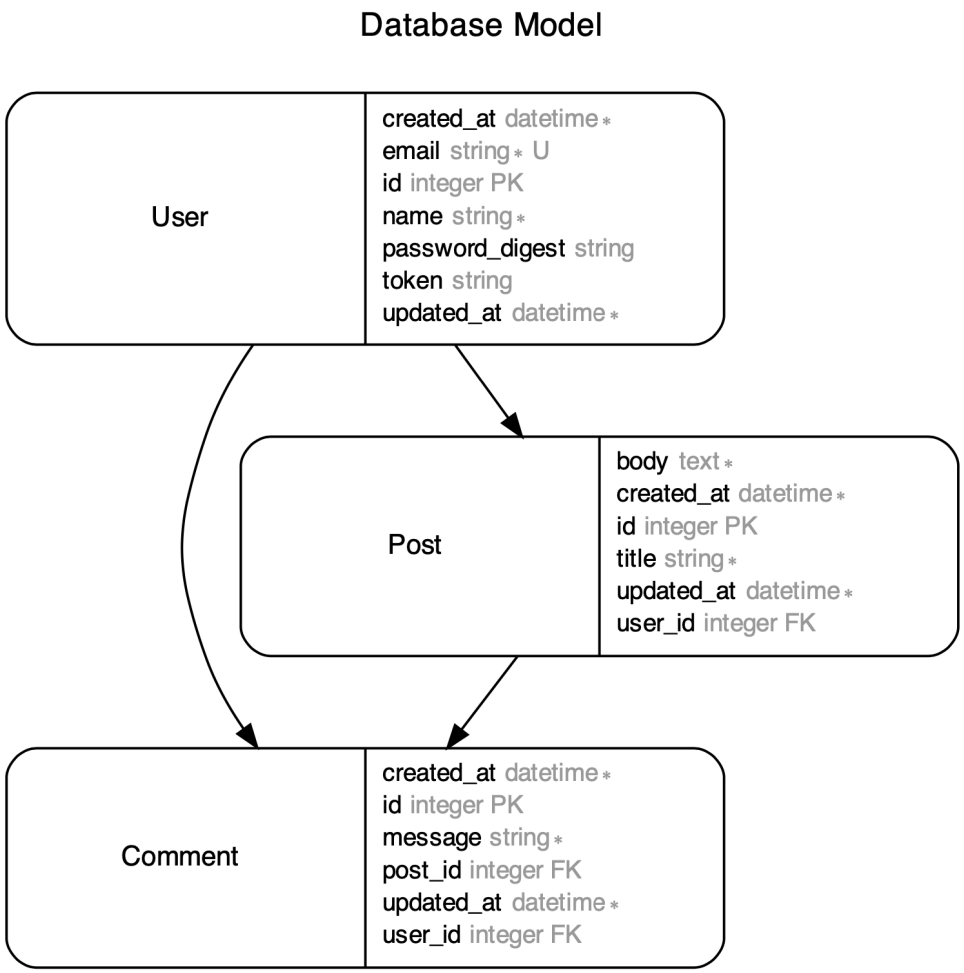

Figura 6.7: Arquitetura do banco de dados do sistema de blog.

Na Figura 6.7 podemos ver que o banco de dados é mais robusto que no teste anterior. O banco possui 3 tabelas, "User", "Post", "Comment". Todas as 3 tabelas possuem as colunas "id"que é a chave primária, e as colunas "created_at"e "updated_at"que representam a data de criação e da última edição dos registros. A tabela "User"possui um campo de email e de nome, um campo chamado "password_digest"que representa a senha criptografada por uma função hash, e um campo chamado "token"que representa uma sequencia aleatória de caracteres que definem a sessão de um usuário.

A tabela "Post"se refere às postagens, ela possui uma chave estrangeira "user_id"de maneira que cada usuário possui muitas postagens. Ela possui outras duas colunas, "title"e "body", que representam respectivamente o título e o texto da postagem.

Por último temos a tabela "Comment"que se refere a tabela de comentários. Ela possui duas chaves estrangeiras, "user_id"e "post_id", representando que um usuário e uma postagem possuem muitos comentários. Além desses campos, a tabela possui uma coluna "message"que representa o próprio texto do comentário.

Regras de negócio

O sistema de blogs possui muitas regras de negócios, listadas a seguir:

- Para a criação de um usuário, os campos "password", "password_confirmatin", "email", "name"não podem ser vazios.
- O "email"precisa ser único e possuir um formato esperado de um email.
- A senha precisa ter no mínimo 6 caracteres.
- Para definir uma senha, "password"precisa ser igual a "password_confirmation".
- Um usuário pode editar e deletar apenas a si mesmo, e somente quando autenticado (feito o login).
- As colunas "token"e "password_digest"não são retornadas junto do usuário na resposta da requisição.
- Na tabela "Post", as colunas "title"e "body"não podem ser vazias.
- Apenas um usuário autenticado pode criar uma postagem.
- Uma postagem pode ser editada ou excluída, apenas por um usuário autenticado que seja o mesmo que criou a postagem.
- Na tabela "Comment"a coluna "message"não pode ficar em branco.
- Apenas um usuário autenticado pode criar um comentário.

- Um comentário apenas pode ser editado ou excluído, por um usuário autenticado que seja o mesmo que criou o comentário.
- Toda vez que um comentário ou uma postagem for retornada em uma requisição, deve ser retornado, junto no JSON, o usuário criador dos mesmos.

API's do sistema

O sistema de blog possui um total de 16 API's, e considerando que o sistema está sendo executado localmente na porta 3000, temos os seguintes endpoints disponíveis para serem acessados no sistema:

```
API para acessar todos os usuários existentes:
GET http://localhost:3000/users

--------

API para criar um novo usuário:
POST http://localhost:3000/users
Body:
[user]name                      exemplo_de_nome
[user]email                     exemplo_de_email
[user]password                  exemplo_de_senha
[user]password_confirmation     exemplo_de_senha

--------

API para editar o usuário autenticado:
PATCH http://localhost:3000/user
Header:
Authorization                   token_do_usuário
Body:
[user]name                      exemplo_de_nome
[user]email                     exemplo_de_email
[user]password                  exemplo_de_senha
[user]password_confirmation     exemplo_de_senha

--------
```

```
API para acessar um usuário em específico:
GET http://localhost:3000/users/:user_id

--------

API para acessar o usuário autenticado:
GET http://localhost:3000/user
Header:
Authorization                   token_do_usuário

--------

API para excluir o usuário autenticado:
DELETE http://localhost:3000/user
Header:
Authorization                   token_do_usuário

--------

API para recuperar todas as postagens feitas por um usuário:
GET http://localhost:3000/users/:user_id/posts

--------

API para realizar o login:
POST http://localhost:3000/auth/login
Body:
email                           exemplo_de_email
password                        exemplo_de_senha

--------

API para acessar todas as postagens criadas:
GET http://localhost:3000/posts

--------
```

API para acessar uma postagem específica:
GET http://localhost:3000/posts/:post_id

API para excluir uma postagem:
DELETE http://localhost:3000/posts/:post_id
Header:
Authorization token_do_usuário

API para criar uma postagem:
POST http://localhost:3000/posts
Header:
Authorization token_do_usuário
Body:
[post]title exemplo_de_nome
[post]body exemplo_de_email

API para editar uma postagem:
PATCH http://localhost:3000/posts/:post_id
Header:
Authorization token_do_usuário
Body:
[post]title exemplo_de_nome
[post]body exemplo_de_email

API para acessar todos os comentários de uma postagem:
GET http://localhost:3000/posts/:post_id/comments

```
API para excluir um comentário:
DELETE http://localhost:3000/posts/:post_id/comments/:comment_id
Header:
Authorization                   token_do_usuário

--------

API para criar um comentário:
POST http://localhost:3000/posts/:post_id/comments
Header:
Authorization                   token_do_usuário
Body:
[comment]message                exemplo_de_texto
```

6.2.2 Testes manuais

Através da ferramenta Postman [27] realizamos os seguintes testes manuais:

- Criar alguns usuários com os parâmetros corretos, Figura 6.8.
- Tentar criar um usuário enviando parâmetros incorretos, esperando um erro.
- Editar o usuário autenticado com os parâmetros corretos.
- Tentar editar o usuário autenticado enviando parâmetros incorretos, esperando um erro.
- Tentar editar um usuário sem estar autenticado, esperando um erro.
- Acessar a lista de todos os usuários.
- Acessar um usuários específico pelo id.
- Deletar o usuário autenticado.
- Tentar excluir um usuário sem estar autenticado, esperando um erro.
- Realizar um login corretamente.
- Tentar realizar o login enviando uma senha e um email inválidos para se obter um erro.
- Obter o usuário autenticado.
- Obter todos as postagens que um usuário possui.

- Criar uma postagem com os parâmetros corretos.
- Tentar criar uma postagem enviando parâmetros incorretos, esperando um erro.
- Tentar criar uma postagem sem estar autenticado, esperando um erro.
- Editar uma postagem com os parâmetros corretos.
- Tentar editar uma postagem enviando parâmetros incorretos, esperando um erro.
- Tentar editar uma postagem sem estar autenticado, esperando um erro.
- Tentar editar uma postagem feita por outro usuário, esperando um erro.
- Excluir uma postagem com os parâmetros corretos.
- Tentar excluir uma postagem sem estar autenticado, esperando um erro.
- Tentar excluir uma postagem feita por outro usuário, esperando um erro.
- Acessar todos as postagens criadas.
- Acessar alguma postagem específica.
- Acessar todos os comentários de uma postagem.
- Criar um comentário com os parâmetros corretos.
- Tentar criar um comentário enviando parâmetros incorretos, esperando um erro.
- Tentar criar um comentário sem estar autenticado, esperando um erro.
- Excluir um comentário com os parâmetros corretos.
- Tentar excluir um comentário sem estar autenticado, esperando um erro.
- Tentar excluir um comentário feito por outro usuário, esperando um erro.

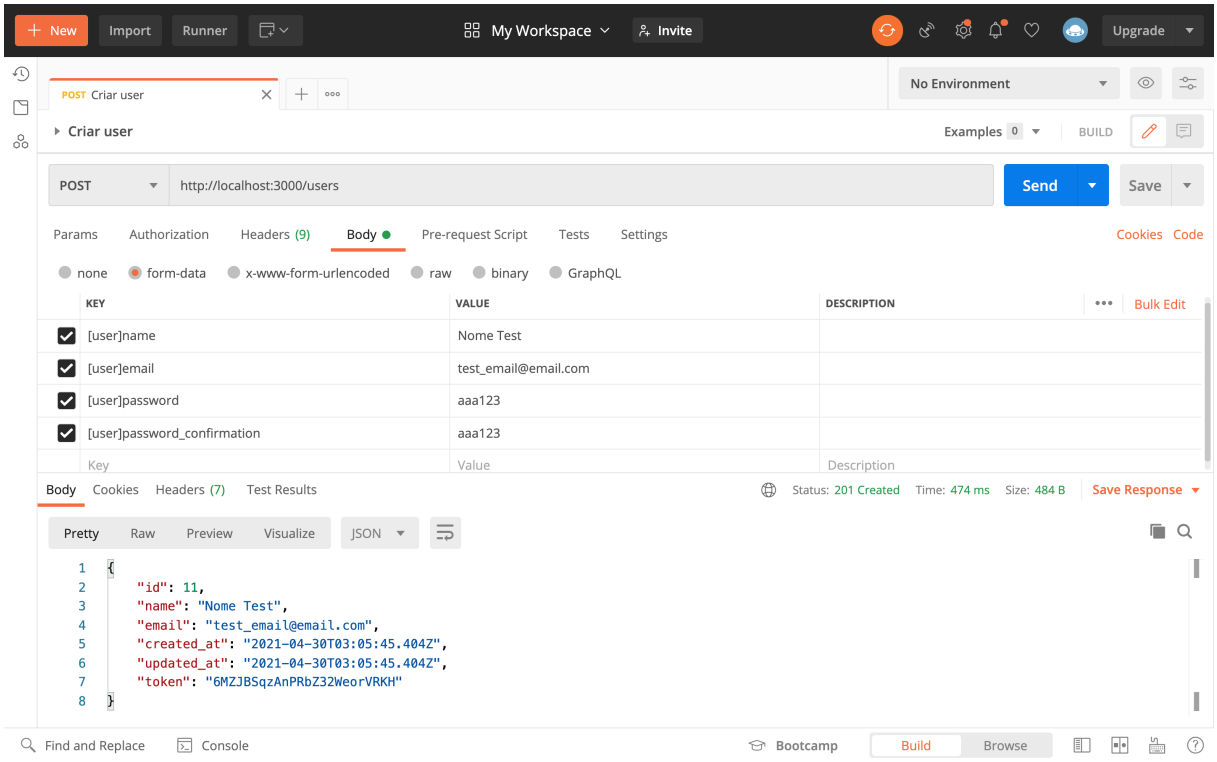

Figura 6.8: Exemplo de um cadastro de usuário feito manualmente através da ferramenta Postman.

6.2.3 Uso da gema Regression Generator

Após realizar os testes manuais listados acima com a gema Regression Generator ativada, foi criado um arquivo de logs conforme mostrado na Figura 6.9 em que mostra um log de duas postagens sendo recuperadas pelo id e tendo seus campos atualizados. Ao executar o comando:

```
rails regression_generator:build
```

Todos os testes automatizados são gerados. Pode-se observar um fragmento desses testes na Figura 6.10, em que é apresentado 4 cenários de testes:

- Cenário de falha ao se criar uma conta de usuário com um email que já existe em outra conta.
- Cenário de recuperação de todos os usuários cadastrados no sistema.
- Cenário de sucesso de uma autenticação por email e senha.
- Cenário de falha de uma autenticação por email e senha ao colocar um email não existente.

Por fim, com os testes prontos pode-se usar o comando "bundle exec cucumber"para que a ferramenta Cucumber [18] execute todos os testes automatizados gerados e também utilizar a ferramenta SimpleCov [25] para obter a cobertura de teste do código. Como apresentado respectivamente nas Figuras 6.11 e 6.12, observa-se que nenhum teste falhou e a cobertura alcançada foi de 100%.

Em resumo, os seguintes resultados foram obtidos no primeiro experimento:

- 100% de cobertura de testes.
- 55 cenários de testes criados.
- 377 passos de testes criados.
- Tempo de execução de todos os cenários de teste: 2.127 segundos.
- Tempo necessário para a execução dos testes manuais pela ferramenta Postman: 8 minutos e 36 segundos.

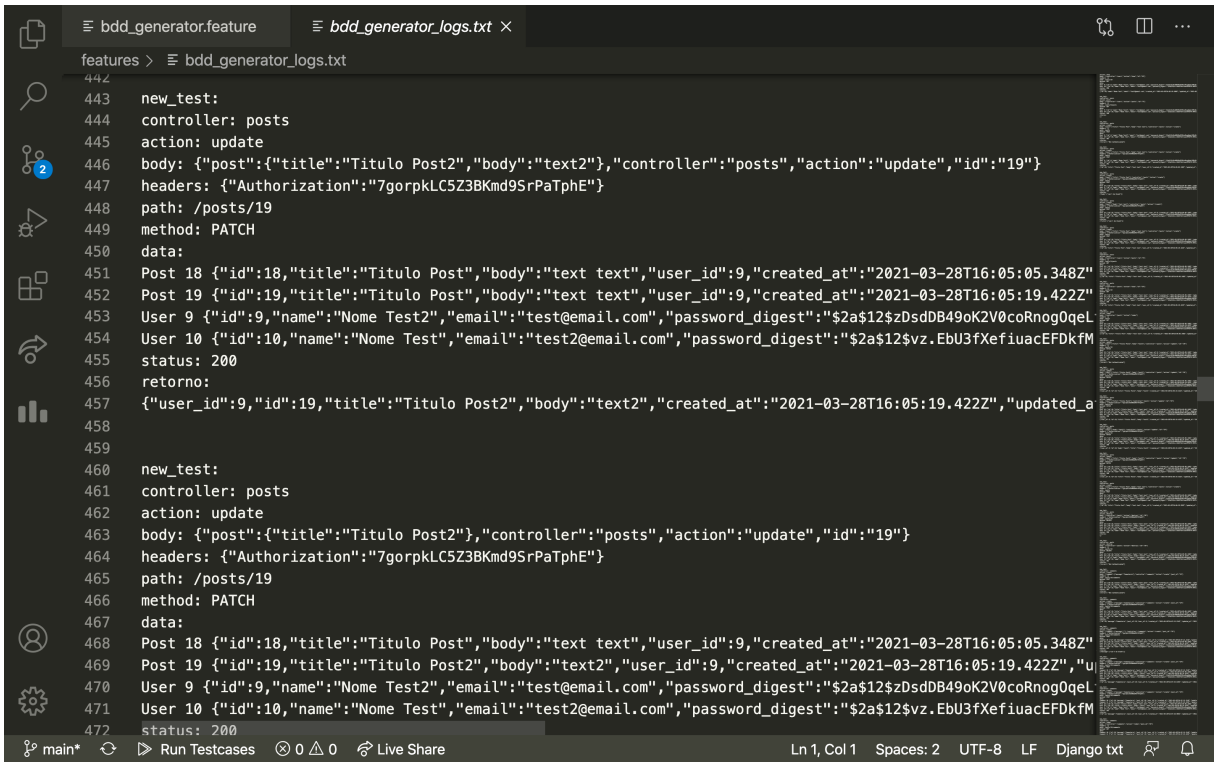

Figura 6.9: Fragmento de logs gerados pela gema Regression Generator no sistema de blog.

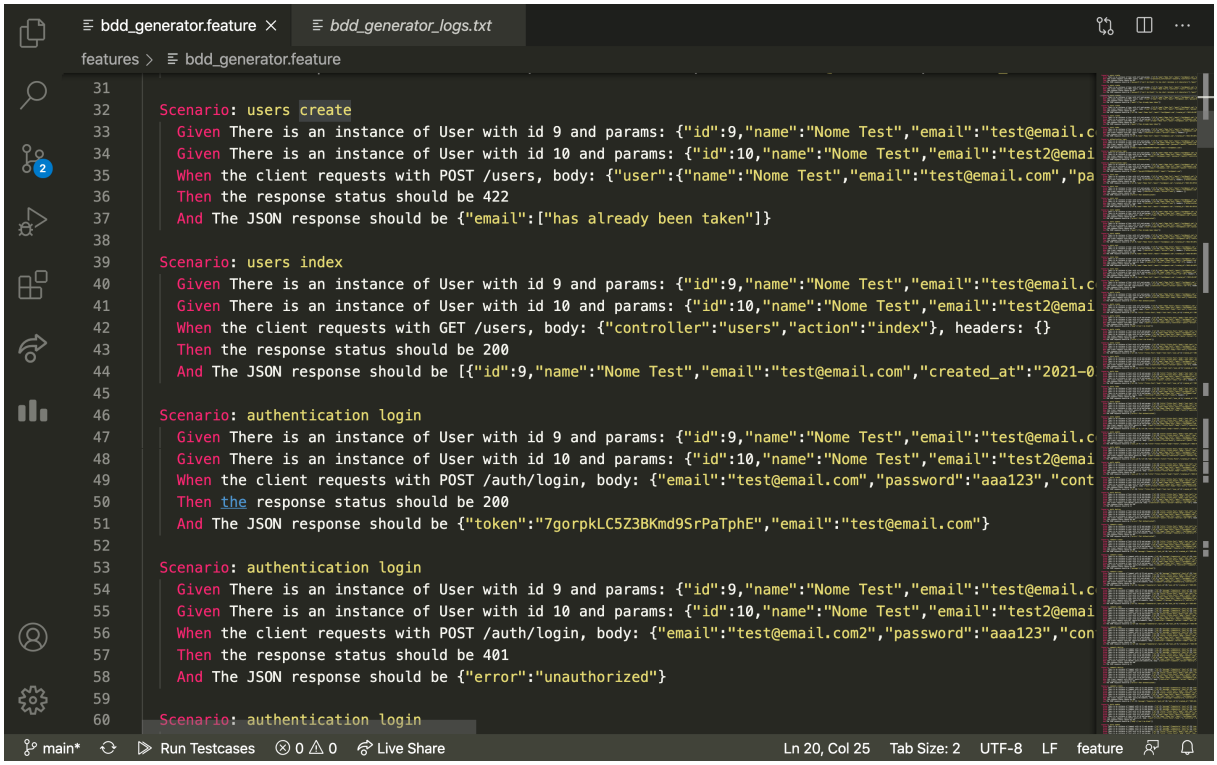

Figura 6.10: Fragmento de testes gerados pela gema Regression Generator no sistema de blog.

Figura 6.11: Execução dos testes gerados pela gema Regression Generator no sistema de blog.

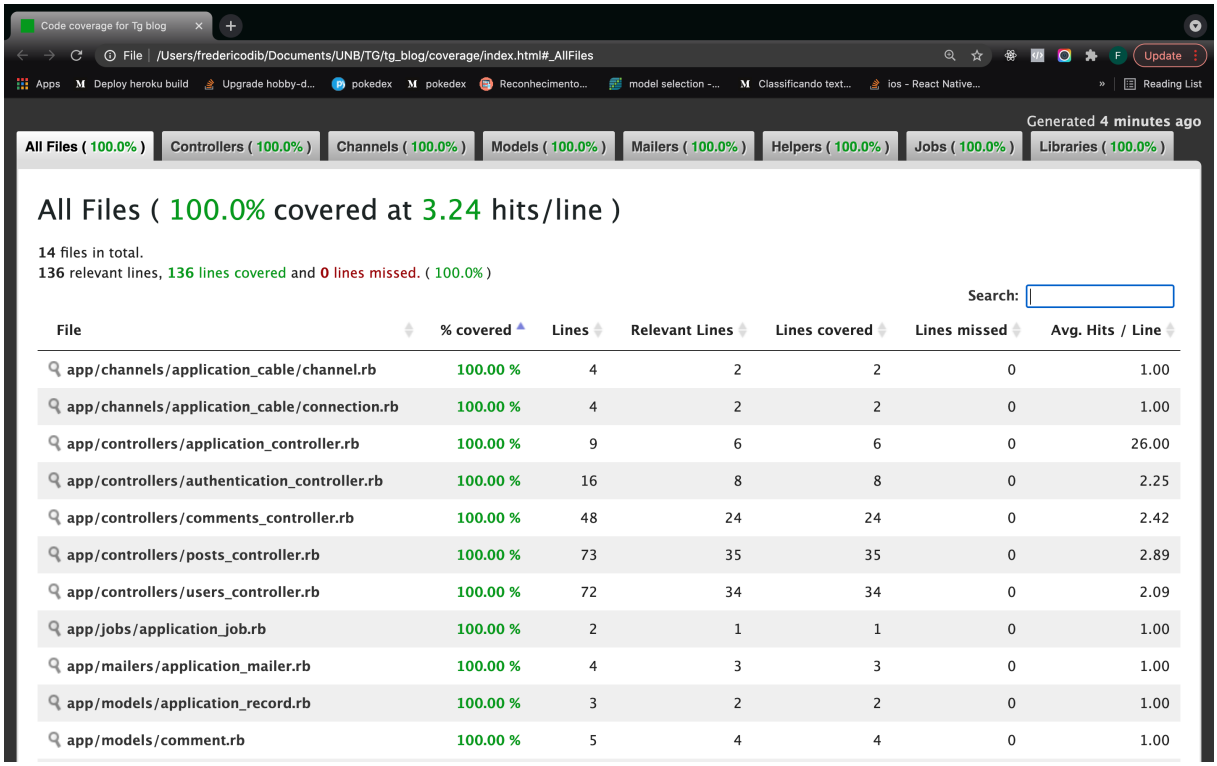

Figura 6.12: Cobertura de testes no sistema de blog.

6.3 Uso no sistema da Sisterwave

A Sisterwave (https://www.sisterwave.com/) é uma startup nascida em Brasília em 2017 e posteriormente acelerada pelo Facebook no segundo semestre de 2019. Ela tem como produto uma rede de apoio para a mulher viajante, que funciona tanto por meio de anfitriãs que disponibilizam algum cômodo em sua residencia para uma sister (usuária do sistema) se hospedar, quanto por meio de chats e grupos para compartilhar dicas e experiências.

Nesta sessão será testada a gema desenvolvida no sistema da Sisterwave. O sistema é bastante grande e possui diversas funcionalidades complexas, como por exemplo:

- Grupos e salas de bate papo.
- Sistema de assinatura.
- Reserva e cancelamento de estadia.
- Localização por mapa.
- Área para gerenciamento dos administradores.
- Entre outros.

Devido ao sigilo do sistema, neste trabalho não poderá ser detalhado as APIs disponíveis e os testes realizados, mas abordaremos o resultado. Após a realização dos testes manuais e da execução do Cucumber, como mostrado na Figura 6.13 foi realizado teste de 36 cenários e 3729 passos de teste, e desses cenários de teste, foi obtido 10 falhas e 26 sucessos. A cobertura final medida pela gema SimpleCov foi de 68% como é possível ver na Figura 6.14.

Os cenários falhados, em grande parte, foram devido a algum tipo de integração com outro sistema que ocorria durante o cenário do teste. Por essas integrações não estarem encapsuladas dentro dos cenários de testes e pelo sistema não ter controle sobre elas, não há como garantir que a condições ocorridas durante a execução do teste manual seja idêntica ao cenário que o teste representa, ocorrendo conflitos de algum id ou código recebido sejam diferentes do esperado. Alguns exemplos desses serviços externos são:

- Serviço de Geolocalização do Google Cloud.
- Serviços de mensageria e norificações.
- Serviço de upload de imagens.
- Serviço de API de pagamento.
- Serviço de Login com o Facebook.

Figura 6.13: Execução dos testes gerados pela gema Regression Generator no sistema da Sisterwave.

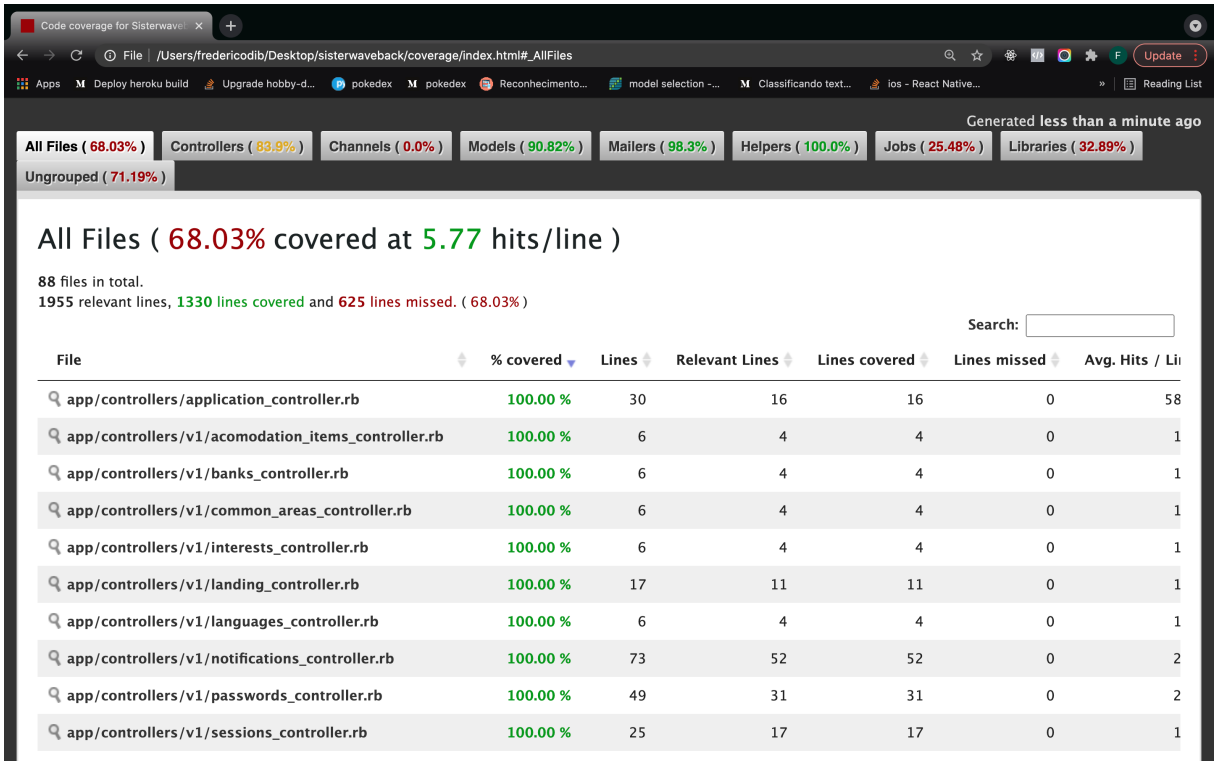

Figura 6.14: Cobertura de testes no sistema da Sisterwave.

6.4 Análise dos Resultados

Como podemos observar na tabela de resultados, Tabela 6.1, a gema obteve resultados muito promissores e satisfatórios. Nos dois primeiros sistemas foi possível obter 100% de cobertura de testes e um número razoável de cenários testados, sendo 19 no sistema de Contatos e 55 no sistema de blogs. No Sistema da Sisterwave que é um sistema mais complexo e possui diversas integrações com outros sistemas, apesar de haver falhas, nos cenários que não foram afetados por outros serviços foi obtido bastante sucesso. Acerca das qualidades dos testes, todos os testes mantiveram seu funcionamento de forma independente uns dos outros e apresentaram rápido tempo de execução, respeitando o principio FIRST.

Sistemas	**Contatos**	**Blog**	**Sisterwave**
Cobertura	100%	100%	68%
Cenários Criados	19	55	36
Passos Criados	86	377	3729
Tempo de execução testes gerados	0.332s	2.127s	49.874s
Tempo de execução testes manuais	2m 33s	8m 36s	46m 23s

Tabela 6.1: Resumo dos resultados

Outros dois ganhos importantes proporcionados pela biblioteca são a persistência dos testes manuais, de maneira que uma vez testado, não haverá necessidade de ser retestado manualmente em futuras incrementações, poupando tempo, e também a velocidade de produção desses testes. Ao passo que o tempo de codificação de 74 cenários de teste levaria muitas horas. Com a gema, no entanto, foi possível construir essa quantidade de testes automatizados em menos de 11 minutos, provendo dessa maneira, um potencial ganho de produtividade.

Capítulo 7

Conclusão

Muitas startups negligenciam os testes automatizados devido ao tempo alto de codificação para gerar esses testes e apenas realizam testes manuais. Por outro lado, com as diversas iterações e incrementos nas funcionalidades do produto, o teste manual precisa ser refeito diversas vezes, se tornando cada vez mais custoso para empresa a longo prazo.

Para solucionar tal problema, no Capítulo 3 foi apresentada a solução da gema Regression Generator e no Capítulo 5 foi apresentada sua implementação. A gema mostrou resultados bastante promissores, apresentando que é possível gerar testes automatizados a partir do que havia sido testado manualmente anteriormente.

A eficiência desses testes gerados pela gema foi evidenciada ao analisar a cobertura de testes nos experimentos realizados, como mostrado no Capítulo 6. Ambos os sistemas, um sistema pequeno e simples e outro mais robusto com diversas funcionalidades, não possuíam nenhum teste, portanto, tinham uma cobertura de 0%. Após realizar os testes manualmente, é possível então gerar testes no formato Gherkin que capturam tais comportamentos, com uma cobertura de 100% em ambos os exemplos avaliados.

Outro resultado que a gema evidenciou por meio das provas de conceito no Capítulo 6 foi a quantidade de testes automatizados gerados em um tempo relativamente curto. No exemplo do sistema de Blog, em 8 minutos de testes manuais foram gerados 55 cenários de testes, o que levariam diversas horas casos um desenvolvedor precisar especificar e implementar os mesmos 55 cenários.

Apesar dos resultados promissores a gema tem muito a ser aprimorada. No momento ainda não existe nenhuma forma intuitiva de selecionar quais cenários deseja-se manter e quais não. A forma como isso é feito atualmente é apagando o cenário indesejado do arquivo de logs gerado, o que é bastante complicado caso o sistema possua um grande número de cenários de testes. Outro grande problema, como mostrado no Capítulo 6, são em casos de sistemas mais robustos, que apresente integrações com muitos outros sistemas, os testes gerados pela gema estão sujeitos a falhas. Os trabalhos futuros envolvem a criação

de uma interface gráfica na qual o desenvolvedor possa gerenciar seu testes, dividi-los em categorias e remover os cenários de testes indesejados ou similares e também desenvolver alguma ferramenta que permita lidar com integrações externas.

Referências

[1] Anand, Azeem Uddin1 Abhineet: *Importance of software testing in the process of software development*. IJSRD - International Journal for Scientific Research Development, 6(12):141–145, 2019. 1

[2] Márcio Eduardo Delamaro, José Carlos Maldonado, Mario Jino: *INTRODUÇÃO AO TESTE DE SOFTWARE*, volume 4. Elsevier Editora Ltda, 2007. 1

[3] Semedo, Maria João Moreno: *Ganhos de produtividade e de sucesso de Metodologias Ágeis VS Metodologias em Cascata no desenvolvimento de projectos de software*. Master's thesis, Universidade Lusófona de Humanidades e Tecnologias, 2012. 1

[4] Giardino, C., Unterkalmsteiner M. Paternoster N. Gorschek T. Abrahamsson P: *What do we know about software development in startups?* IEEE software, 31(5):28–32, 2014. 2

[5] Guide, Ruby: *Ruby gems*. `https://www.ruby-lang.org/pt/libraries/`. 3, 18, 24

[6] Rails Guide, Ruby on: *Ruby on rails*. `https://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark`, acesso em 2021-01-18. 3, 9, 11, 15, 18, 22, 24, 33, 35

[7] RedHat: *O que é api?* `https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces`. 3

[8] Castro, Vinicius Almeida: *Introdução ao desenvolvimento Ágil*. `https://www.devmedia.com.br/introducao-ao-desenvolvimento-agil/5916`, acesso em 2007. 4

[9] Kent Beck, James Grenning, Mike Beedle: *Manifesto Ágil*. `http://agilemanifesto.org/iso/ptbr/manifesto.html`. 4

[10] North, Dan: *O que é json*. `https://www.hostinger.com.br/tutoriais/o-que-e-json`, acesso em 2020. 6, 14, 19

[11] Duggal, Gaurav e Mrs Bharti Suri: *B.suri: —understanding regression testing techniques*, 2008. 7

[12] Fox, Armando e David Patterson: *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC, 2021. 8, 9

[13] Matsumoto, Yukihiro "Matz": *Ruby*. https://www.ruby-lang.org/pt/about/. 9, 18, 21, 24

[14] Grady Booch, Robert A. Maksimchuk, Michael W. Engle: *Object-oriented analysis and design with applications*. Addison-Wesley Professional, 2007. 10

[15] *What is aspect-oriented programming*. https://docs.jboss.org/aop/1.0/aspect-framework/userguide/en/html/what.html, acesso em 2020-03-03. 10

[16] Pop, Dragos Paul e Adam Altar: *Designing an mvc model for rapid web application development*. Procedia Engineering, 69(25):1172–1179, 2014. 15, 19

[17] Cucumber: *Gherkin reference*. https://cucumber.io/docs/gherkin/reference/. 17, 19, 29

[18] Cucumber: *Cucumber*. https://cucumber.io/. 17, 19, 20, 21, 24, 29, 31, 33, 35, 38, 51

[19] Guide, Ruby: *Ruby gems*. https://rubygems.org/. 18

[20] Facebook: *React, uma biblioteca javascript para criar interfaces de usuário*. https://pt-br.reactjs.org/. 19

[21] Cucumber: *Cucumber-rails*. https://github.com/cucumber/cucumber-rails. 19

[22] Rspec: *Behaviour driven development for ruby. making tdd productive and fun*. https://rspec.info/. 20, 21

[23] Wampler, Dean: *Aquarium gem*. https://github.com/deanwampler/Aquarium. 20, 24, 25

[24] Tagwerker, Ernesto: *Database cleaner*. https://github.com/DatabaseCleaner/database_cleaner. 21, 30

[25] *Simplecov*. https://github.com/simplecov-ruby/simplecov. 21, 35, 39, 51

[26] Docs, Mozilla Web: *Crud*. https://developer.mozilla.org/pt-BR/docs/Glossary/CRUD. 27

[27] Postman: *Postman, the collaboration platform for api development*. https://www.postman.com/. 35, 37, 48

[28] *Github*. https://github.com/. 35, 43