Um estudo sobre Técnicas de Teste de *Software* no *Framework Laravel*

Angelica Caetane Pelizza[1], Cristiano Bertolini[1], Sidnei Renato Silveira[1]

[1]Departamento de Tecnologia da Informação (UFSM)
Caixa Postal 54 – 98.400-000 – Frederico Westphalen – RS – Brasil

angelicapelizza@hotmail.com, cristiano.bertolini@gmail.com,
sidneirenato.silveira@gmail.com

Resumo. *Frameworks no estilo* Model-View-Controler *são úteis para acelerar e melhorar o desenvolvimento de software. No entanto, o teste de software continua sendo necessário, e muitas vezes é deixado de lado, pelo fato de se usar um framework de desenvolvimento. Neste contexto, este trabalho apresenta um estudo referente aos tipos e técnicas de teste de software de acordo com os padrões existentes no framework Laravel, visando proporcionar maior qualidade no processo de desenvolvimento de software. Com base na análise das técnicas e as funcionalidades de teste que o Laravel oferece, foram estudados e desenvolvidos esquemas de teste para o Framework Laravel.*

Palavras-Chave: *Teste de Software, Framework Laravel, MVC*

Abstract. *Frameworks in the* Model-View-Controller *style are useful for accelerating and improving software development. However, continuous software testing is needed, and is often overlooked by the fact of using a development framework. In this context, this paper presents a study of types and techniques for Software testing according to existing standards in the Laravel framework, aiming at the source of large-sized and effective development process the software. Based on the analysis of the test types and as test functionalities of Laravel offers, test schemes have been developed, according to the Laravel Framework.*

Keywords: *Software testing, Framework Laravel, MVC*

1. Introdução

Sistemas dependentes de *Software* estão cada vez mais presentes em nosso dia a dia, e podem ser utilizados em diferentes idiomas, sistemas operacionais e plataformas. Um exemplo é a Internet das Coisas *(IoT – Internet of Things)*, em que objetos são participantes ativos de processos, por meio de uma infraestrutura de rede dinâmica e global que atribui competências de autoconfiguração, fundamentadas em protocolos de comunicação padronizados e com capacidade de se comunicar de forma transparente. Esses objetos se tornam participantes ativos em diversos processos e são capazes de interagir de forma autônoma com os eventos do mundo físico real, possuindo capacidades para intervir em determinadas situações sem a necessidade de intervenção humana [Teixeira et al. 2014].

Desta forma, o *software* possui um papel importante no cotidiano de diversos grupos de pessoas, dispondo de diversas opções de forma eficaz, ou até mesmo, podendo ser frustrante caso não cumpra os requisitos desejados. Para garantir a qualidade de um *software* é utilizado um processo, ou uma série de processos, que tem a finalidade de analisar se o que o código faz foi determinado para que ele faça, ou seja, revelar a presença de erros no *software* [Myers et al. 2011]. Testes bem sucedidos determinam casos de teste para os quais o programa em teste falhe, visando garantir que eles não tenham defeitos.

É de suma importância a aplicação das atividades de teste no processo de desenvolvimento de um *software*, dispondo de uma oportunidade para correção de eventuais problemas antes da entrega ao usuário final [Pressman 2005]. Considerando um ambiente cada vez mais dinâmico aos *softwares*, observa-se que se exige cada vez mais atenção dos desenvolvedores e testes mais complexos para que seja mantida a confiança, a funcionalidade e o desempenho do sistema, o que torna o uso de testes manuais menos aplicável, tanto por questões econômicas quanto práticas [Leicht et al. 2017]. Outra situação é que as organizações sofrem pressão para o desenvolvimento de sistemas de informação em um curto período de tempo, independente de serem integrados com outros sistemas ou não.

Tem-se uma diversidade de casos de teste, então é necessário analisar quais são mais adequados para uma determinada situação, a fim de obter o melhor resultado e custo. De acordo com Wong [Wong et al. 1995] há três elementos básicos para classificar e adequar os critérios de teste, que são: o custo, a eficácia e a dificuldade de satisfação. No custo tem-se a quantidade de casos de teste que foram necessários e o tempo necessário para a sua execução, o aprendizado para a geração de casos de teste e os casos que foram realizados manualmente. A eficácia diz respeito à capacidade de um determinado tipo de teste encontrar um maior número de defeitos que outro. Por fim, a dificuldade de satisfazer um critério tendo satisfeito outro.

Bastos [Bastos et al. 2012] elencam os processos de testes dentro das empresas, e expõem a maneira como são executados esses processos. Em sua maioria, ocorrem pelos próprios desenvolvedores e possíveis usuários do sistema, não sendo suficientes para garantir a qualidade, sendo necessária de realização de testes tidos como um processo organizado, paralelo e integrado ao processo de desenvolvimento, a fim de reduzir os custos de manutenção. Quanto mais tarde um defeito é detectado, o custo para a correção tende a aumentar. Um exemplo é a Regra 10 de Myers [Myers et al. 2011] que afirma que os testes unitários (i) removem erros em um percentual entre 30% a 50% dos defeitos dos programas; (ii) testes de sistemas podem remover entre 30% e 50% dos defeitos remanescentes, utilizadas as técnicas, ainda vão para a produção com aproximadamente 49 % de defeitos, e as revisões finais de código podem reduzir entre 20% e 30% dos defeitos restantes.

Neste contexto, neste trabalho desenvolveu-se um estudo a cerca dos tipos de teste de *software* e ferramentas de teste de *software* disponíveis no *framework Laravel*. Foi proposta a integração dos tipos de teste existentes no *framework Laravel* com técnicas de teste, visando, além da análise dos tipos de teste disponíveis, eficiência e eficácia no processo de teste. Também verificou-se os tipos especiais de teste que o framework Laravel fornece, como a parametrização de dados que, além de facilitar o desenvolvimento de software em relação aos relacionamentos do banco de dados, facilita o processo de

automação de teste de software.

Para dar conta desta proposta, este trabalho está organizado da seguinte forma: a Seção 2 apresenta um breve referencial teórico composto por conceitos de Teste de Software e os tipos de teste; a Seção 3 apresenta o estado da arte destacando alguns trabalhos relacionados; a Seção 4 descreve a o estudo de caso realizado. Por fim, são apresentadas as conclusões e as referências empregadas.

2. Referencial Teórico

Esta seção apresenta o referencial teórico que inclui os principais conceitos de teste de *software* e os tipos de teste.

2.1. Teste de Software

O desenvolvimento de *software* requer que sejam aplicadas diversas técnicas e ferramentas para garantir a qualidade do desenvolvimento do *software*, com o objetivo de minimizar a ocorrência de erros e riscos associados. O teste diz respeito a uma análise dinâmica do produto e é uma atividade fundamental para que sejam identificados os erros. Os testes de *software* são relevantes em execuções de depurações, manutenção e na estimativa de confiabilidade do *software*. Há duas técnicas em destaque que podem ser usadas como base para os testes: (i) teste estrutural, também conhecido como teste de caixa branca e (ii) teste funcional, também conhecido por teste de caixa preta.

No **teste estrutural**, os aspectos de implementação são muito importantes na escolha dos casos de teste e são baseados no conhecimento da estrutura de implementação interna. A maior parte dos critérios desta técnica, utilizam uma apresentação de programa conhecida como grafo de fluxo de controle ou grafo de programa, onde um determinado programa é decomposto em blocos separados e os mesmos são executados na ordem em que lhes foi determinada [Maldonado et al. 2004].

No **teste funcional** pode-se visualizar apenas o lado externo, sendo voltado aos dados de entrada e às respostas produzidas, consideradas saídas. Nele verificam-se funções do sistema sem considerar os detalhes da implementação de código. Os dois passos principais do teste funcional são identificar as funções que o *software* está realizando e criar casos de teste capazes de avaliar se essas funções estão de acordo com o resultado esperado. É primordial que se tenha uma especificação bem estruturada e de acordo com os requisitos solicitados pelo usuário [Maldonado et al. 2004].

Para definir o momento em que a execução dos testes deve ser cessada, e que a ocorrência de defeitos não irá compensar o aumento do custo dos testes, utilizam-se dois termos: *under test* e *over test* (falta de teste e excesso de teste). Assim, prever o momento ideal para que os testes sejam concluídos requer um planejamento detalhado e que seja seguido com precisão. Uma recomendação genérica é que os testes de *software* devem ser interrompidos quando o intervalo de ocorrência de defeitos começa a aumentar muito (de horas para dias) [Bastos et al. 2012].

2.2. Tipos de Teste

Nesta seção serão apresentados os principais tipos de teste. A Tabela 1 apresenta os tipos de teste e seus respectivos níveis. A maior parte dos testes são baseados em três fases: teste de unidade, de integração e de sistema [Pressman 2005].

Tabela 1. Tipos e Níveis de teste

Tipo de Teste	Nível
Teste de Unidade	Código (verificação em nível de módulos)
Teste de Integração	Projeto (análise do comportamento esperado)
Teste de Sistemas	Requisitos (análise da compatibilidade dos recursos)

2.2.1. Teste de Unidade

No teste de unidade ocorre a verificação sobre a menor unidade de projeto, para verificar a lógica e a implementação em cada um dos módulos, sendo orientado pela técnica estrutural, que se insere diretamente sobre o código-fonte dos componentes estruturais para tratar eventuais problemas em relação à condição, fluxo de dados, teste de ciclos e teste de caminhos lógicos. Estes componentes e módulos são testados, geralmente, pelos próprios desenvolvedores por serem pequenas unidades [Pressman 2005]. Quando os testes unitários são ignorados e são usados para teste em blocos maiores, já integrados com outros componentes, tendem a ocorrer transtornos uma vez que, caso haja um erro, há um espaço maior para encontrá-lo e para determinar e consertar a causa deste erro [Silva 2008].

Outro ponto a ser considerado é o Desenvolvimento Orientado a Teste citado por Way [Way 2013], um padrão de *software* ágil utilizado pelos desenvolvedores, onde se desenvolve um teste antes de desenvolver o código em si. Possui três regras primordiais: i) escrever um teste de falha para cada linha de código; ii) escrever a quantidade mínima de código para fazer o teste passar e definir perspectivas, como por exemplo, "qual é a maneira mais simples de fazer esse teste passar?"; iii) após os testes serem aplicados e validados, pode-se refatorar o código.

2.2.2. Teste de Integração

O teste de integração propõe uma técnica sistemática para a construção da estrutura do programa, verificando os erros que podem estar associados à interface, visando contribuir para uma estrutura ideal ao projeto [Pressman 2005]. Considerando um sistema baseado em componentes, onde os mesmos trocam informações com o propósito de compor as funcionalidades do sistema, e supondo que o teste unitário já tenha sido aplicado, nesta fase é analisada a sua integração para verificar se o sistema está se comportando do modo esperado. As estratégias mais utilizadas neste caso são a *bottom-up* e *top-down*.

A *bottom-up* inicia o teste no mais baixo nível a ser testado chegando ao mais alto nível; já no *top-down* ocorre o oposto, o sistema é testado do alto nível para o mais baixo nível. Considera-se o programa como um componente único que possui sub-componentes denominados *stubs*. Um *stub* simula o comportamento de um componente ainda não integrado no sistema, que possui a mesma interface mas funcionalidades limitadas, sendo assim testados e implementados de acordo com a estratégia determinada [Gouveia 2004].

Dentre os testes de integração, é citada a integração contínua, que objetiva mesclar as revisões diárias e as revisões fonte, evitando, de certa forma, o caos que uma integração pode gerar.

2.2.3. Teste de Sistema

Depois de analisadas as funcionalidades de forma isolada e integrada, é aplicado o teste de sistema, que visa analisar se os elementos do sistema foram integrados de maneira correta e se suas funções estão de acordo com a especificação [Pressman 2005]. É realizada a execução de acordo com a perspectiva dos usuários finais, aplicando situações semelhantes que envolvam ambiente, volumes de dados e interfaces gráficas. Quanto mais semelhantes às condições de uso estiverem das reais, mais confiável dar-se-á o teste de sistema [Silva 2008].

A finalidade de um teste de sistema é analisar o *software*, buscando falhas, por meio da sua utilização, simulando o comportamento de um usuário final, sendo executado no mesmo ambiente, sob as mesmas condições de uso. Também são utilizadas as possíveis entradas que um usuário poderia vir a utilizar no cotidiano [Neto and Arilo 2006].

Além destes, citam-se os testes de esforço, que são feitos para avaliar o comportamento de um sistema em situações que exigem recursos em quantidade, frequência ou volume elevados, buscando combinações de dados e/ou entradas válidas que podem causar instabilidade ou processamento inadequado com eventuais falhas [Pressman 2005].

3. Estado da Arte

Nesta seção apresentam-se alguns trabalhos relacionados ao estudo desenvolvido. Foram analisados trabalhos referentes aos tipos de teste e ferramentas de teste de *software* em uma arquitetura MVC.

3.1. Pesquisa de testes de software: realizações, desafios, sonhos

Bertolino [Bertolino 2007] apresenta os testes de *software* como uma atividade essencial na Engenharia de *Software*. Testes são amplamente utilizados para a garantia da qualidade, examinando um *software* em desenvolvimento até a sua execução, dispondo de um *feedback* real em relação a sua execução.

Considerando que os testes envolvem todo o grupo de desenvolvedores, diversas atividades, técnicas e um planejamento detalhado, apresentando desafios complexos. Esses fatores fazem com que os testes se tornem mais custosos e difíceis. Além disso, leva-se em consideração a constante mudança em relação à evolução dos sistemas e aos impactos econômicos de um plano de testes inadequado. Considerada a abrangência do termo teste de *software* (teste unitário, de integração, de aceitação, etc.), podem ser definidos requisitos diferentes em cada etapa, dependendo da aplicação do *software* em teste. Esta variedade de requisitos e objetivos gera desafios, sendo possível retirar de cada etapa do teste o que é comum entre os diferentes tipos de testes.

Assim, é proposta uma amostra de execuções que caracterizam uma visão geral do teste de *software*, classificadas como: (i) por que (por que são feitas observações? é necessário avaliar a usabilidade da interface? o produto pode ser lançado?); (ii) como (como pode ser feita a seleção do teste? quais técnicas serão aplicadas?); (iii) quanto (qual será o tamanho da amostra?) (iv) o que (o que foi executado?) (v) onde (onde será realizada a observação em relação à execução?) (vi) quando (qual o ciclo de vida do produto?). Com estas definições pode-se obter uma visão ampla do que será testado para uma melhor organização de um roteiro de teste.

Este trabalho apresenta a importância do planejamento de teste de software utilizando metodologias disponíveis que sejam ágeis para testar as falhas presentes no *software*. Nosso trabalho apresenta as técnicas disponíveis, funcionalidades e o modelo de desenvolvimento no *framework Laravel*.

3.2. Análise baseada em navegadores de aplicativos de *Framework* da *Web*

A constante evolução e, consequentemente, a sofisticação dos aplicativos *web*, trouxe inúmeros benefícios, tais como a interação dinâmica, atualizações parciais da página, entre outros. Entretanto, a desvantagem é a complexidade gerada em relação ao teste e à localização dos erros. Fundamentando-se disto, de modo geral, o fluxo de aplicação ao utilizar quadros *web*, onde diversas solicitações são processadas pela estrutura da *web* para que se possa disponibilizar páginas dinâmicas que são requeridas e apresentadas ao cliente. É um processo considerado complexo ao se considerar os tipos de solicitação, os parâmetros e o estado de sessão, uma vez que podem resultar em caminhos diferentes do esperado. Estes fatores influenciam diretamente sobre os testes e aplicativos *web* e, os testes de navegador detectam apenas os erros e não sua causa [Kersten and Goedicke 2010].

Selecionados os elementos solicitação/resposta, e visando diminuir o tempo para encontrar o caminho preciso em que ocorre o erro, propõe-se que, quando selecionado um elemento dentro da página gerada, sejam obtidas informações sobre o código fonte e as partes utilizadas durante o processamento do lado servidor. Entretanto a ideia não seria possível, por ter processamento unidirecional, além de abranger várias etapas para cada processamento diferente. Consequentemente, utiliza fontes diferentes e esses resultados são perdidos, e a resposta enviada ao usuário não possui informações. Assim é proposto que sejam apresentados metadados relacionados a partir da integração de ferramentas e *frameworks*. Por exemplo, se ocorre um erro com os dados exibidos ao cliente, pode-se ter como resposta os pontos de interesse: atualizações da *model*, dados anexados, fontes ou consultas de dados.

Observa-se que a automação pode facilitar, de diversas formas, a busca de um erro que está ocorrendo no lado cliente. Ferramentas de documentação e as soluções que elas podem apresentar, quando integradas, podem trazer resultados relevantes em diferentes cenários de aplicação. Pode-se realizar uma análise em relação à integração do *Java Server Faces* (JSF) e o quadro *RichFaces* com a sua aplicação no *framework Laravel* para realizar uma etapa do teste de sistema.

Este trabalho apresenta a importancia da integração de ferramentas e a utilização de *frameworks*, a automação do processo de teste visando facilitar a busca pelo erro. Nosso trabalho também apresenta a importância da automação de teste, ma tem foco na automação e ferramentas disponíveis, já integradas, que otimizam a realização de testes no *framework Laravel*.

3.3. Um modelo para testar aplicativos baseados em *Workflow* e MVC

Considerando os *frameworks* de desenvolvimento *web*, em sua maioria fornecem um padrão *web* de desenvolvimento e fornecem diretrizes e restrições a serem seguidas. *Frameworks Web* fornecem blocos de construção, funcionalidades e controle de fluxo entre páginas *web*, e promovem a separação da lógica de negócio. Karam [Karam et al. 2006] propõem um modelo abstrato para testes de aplicativos baseados em MVC e modelo de

fluxo de trabalho, examinando classes e falhas encontradas em aplicações que utilizam *frameworks* MVC e o paradigma de fluxo de trabalho.

Assim, é proposto o modelo gráfico de fluxo de trabalho: *Model-View-Workflow* (MVWf) com base em testes e as metodologias de teste, baseadas em estruturas que podem ser aplicadas no modelo MVC e *web*, baseadas em fluxo de trabalho. Nele tem-se uma hierarquia composta por quatro componentes: fluxo de trabalho (*Workflow*) (wf), *Model* (m), *View* (v), e comportamento e *Layout* (b&h). Os processos consistem em conjuntos de *model* (instruções *Structured Query Language* (SQL) e XML) e *Views* (interação do usuário com a aplicação). Os componentes e *Layouts* são definidos por especificações e restrições comportamentais de todos os campos (definições comportamentais).

Este trabalho propôs um modelo para testar aplicativos baseados em *Workflow* e MVC, que utiliza a arquitetura MVC, com base no fluxo de trabalho detalhando a hierarquia de componentes gerais e conclui que as falhas, nesse tipo de arquitetura, podem ser baseadas no estado e em código. Nosso trabalho tem como base o teste na arquitetura MVC, onde o módulo *Workflow* pode ser considerado semelhante ao módulo *Controller* da arquitetura MVC.

4. Estudo de caso

Neste trabalho foram estudados os de tipos de teste e técnicas de teste de *software* de acordo com os padrões existentes no *framework Laravel*, visando proporcionar maior qualidade e eficácia no processo de desenvolvimento de *software*.

4.1. Framework Laravel

O *Laravel* é um *framework* PHP utilizado para desenvolvimento *web*. Possui uma arquitetura MVC (*Model-View-Controller*) e tem, como aspecto principal, o de auxiliar no desenvolvimento de aplicações seguras e de alto desempenho de forma ágil e simplificada, com código limpo. Outra característica é o incentivo do uso de boas práticas de programação e a utilização de padrões específicos a ele determinados.

Assim como outros *frameworks*, o *Laravel* possui testes já definidos e a estrutura possui métodos auxiliares convenientes que permitem testar expressamente suas aplicações. No diretório de testes podem ser encontrados testes de banco de dados, testes de navegador e testes que geram entradas de dados fictícios. Quando executados os testes, devem ser definidas as variáveis e realizadas as chamadas dos dados que serão testados na aplicação [Laravel 2017].

Segundo Verma [Verma 2014], a arquitetura MVC utilizada pelo *Laravel* é um padrão que visa aumentar a modularidade de sistemas de *software*, sendo apresentada em três camadas:

- *Model*: gerencia os modelos de dados da aplicação, fazendo a interação com o banco de dados, analisando a lógica, os dados e as regras. É uma camada entre os dados e a aplicação, que pode armazenar diversos tipos de dados, de sistemas gerenciadores de bancos de dados, como o MySQL, ou arquivos *eXtensible Markup Language* (XML);
- *View*: representa a camada de interação com o usuário por meio de interfaces. Diz respeito à representação da aplicação *web* e é responsável por mostrar os

dados que a camada *controller* recebe da camada *model*. Pode ser implementado facilmente com o uso do pacote *".blade.php"* ou com código PHP. O *Laravel* inicia sua execução com a extensão de arquivo (.blade.php ou .php) determinando quem deve prosseguir com a execução do modelo;

- *Controller*: recebe as solicitações dos usuários através da *view* para a exibição ou atualização de dados e faz requisições na camada *model* de acordo com a solicitação correspondente. É considerado um *link* entre a *model* e a *view* e dispõe de duas opções de desenvolvimento da lógica: *Router* e *Controller*. Os *Routers* são mais viáveis para páginas *web* estáticas. *Controllers* são definições de escrita para cada página *web*.

A Figura 1[1] representa o fluxo, de forma gráfica, do modelo de arquitetura MVC em um contexto de Internet com uma requisição HTTP e uma resposta em formato *HyperText Markup Language* - HTML ou XML, de forma gráfica.

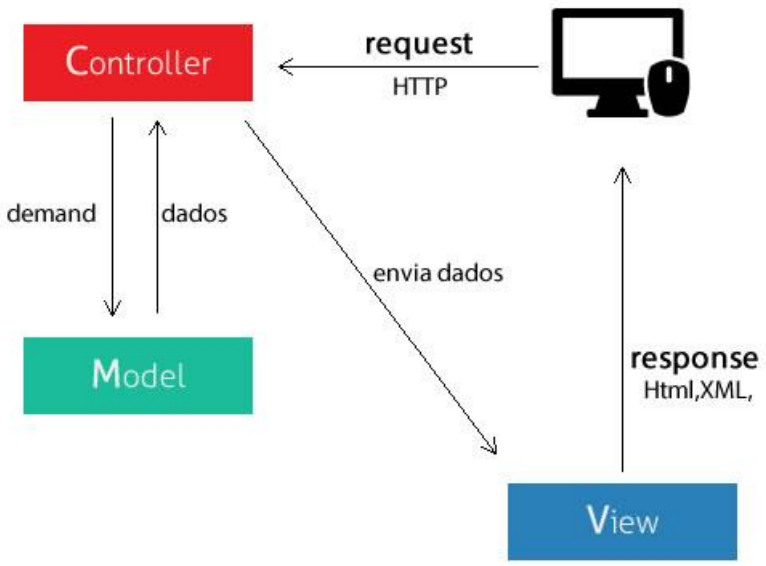


Figura 1. Modelo de arquitetura MVC.

4.1.1. Tipos de Teste no *framework Laravel*

O *framework Laravel* é baseado no *Test-Driven-Development* (TDD), onde é possível obter vantagens em relação à testabilidade do código, uma vez que a estrutura é projetada antes da codificação. Isto evita a repetição de código e códigos confusos, fazendo com que um método possa executar diversas ações. Além disso, fornece integração com a biblioteca de testes unitários *PHPUnit* e possibilita a execução de diversos cenários de teste. O objetivo da utilização do TDD e do *PHPUnit* é facilitar a escrita e a execução de testes unitários no decorrer do desenvolvimento do projeto [Gabardo 2017].

[1]Figura retirada do endereço `https://tableless.com.br/mvc-afinal-e-o-que/`

4.1.2. Testes Unitários

Testes unitários, de acordo com suas definições, compreendem pequenas partes de código, e se tornam mais práticos, de tal maneira que não se repitam processos e o tempo de teste seja otimizado. O *Laravel* possibilita a realização deste tipo de teste, o que faz com que o desenvolvedor obrigatoriamente divida o código (em módulos) de maneira gerenciável para que seja possível testar os módulos individualmente, e também, que o desenvolvedor possua boas práticas em relação à estrutura do código. Um componente importante do teste unitário são as afirmações, que comparam as saídas esperadas com as saídas reais de uma função [Saunier 2014].

Nestes testes, são utilizadas diversas bibliotecas auxiliares para facilitar sua execução. A Figura 2, representa um exemplo de teste simples, que faz uso de duas destas bibliotecas: *WithoutMiddleware*, *DatabaseTransactions* e *DatabaseMigrations*. A biblioteca *WithoutMiddleware* impede que o *middleware* seja executado para esta classe de teste e a *DatabaseTransactions* faz a conexão com o banco de dados, mas neste caso não retorna nenhum valor. Por fim, com a biblioteca *DatabaseMigrations* é definido um ponto de migração caso ocorram alterações para avaliar um antes e depois de cada teste.

Utilizando a pseudo-variável *$this* é possível apontar para um local, neste caso a página principal de uma aplicação inicial simples que o *Laravel* fornece, e visualizar nesta página, uma informação ou campo. Com o auxílio das bibliotecas e funções do PHP é possível executar o teste demonstrado, que visita a página atual e analisa se naquela página está contida a informação que lhe foi solicitada para que ele analisasse, neste caso "Laravel 5".

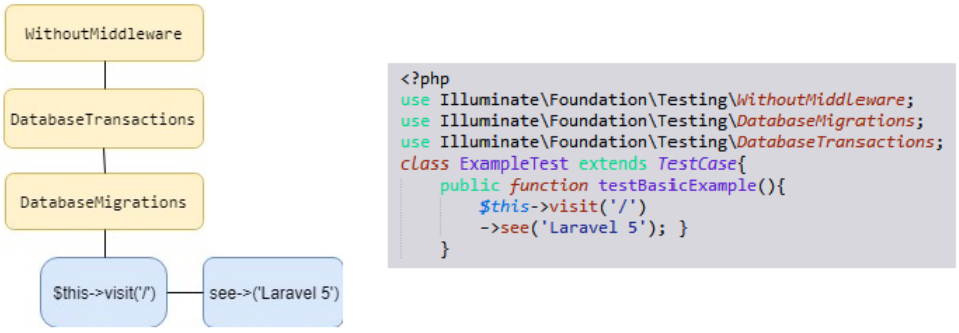


Figura 2. Teste de acesso com uma resposta esperada (Fonte: dos autores, 2017)

A Figura 3 apresenta um teste de autenticação de um usuário comum. Também são utilizadas as bibliotecas citadas anteriormente. Os dados são recebidos por meio da classe *User* e o usuário é criado de maneira fictícia. O método *actingAs()* é responsável por autenticar o usuário determinado como o usuário atual, que será autenticado na aplicação. Os dados podem ser configurados utilizando a matriz do método *WithSession*. Outra maneira mais simples de realizar o teste, é configurar os dados da sessão inicialmente por meio do método *WithSession*, que possibilita obter os dados antes de enviar as solicitações de autenticação para a aplicação. Posterior a isso, é necessário utilizar o método *actingAs()* para a autenticação do usuário. O comando *get('/');* é utilizado, para

que, qualquer modificação que tenha sido realizada no banco de dados seja desfeita após o teste.

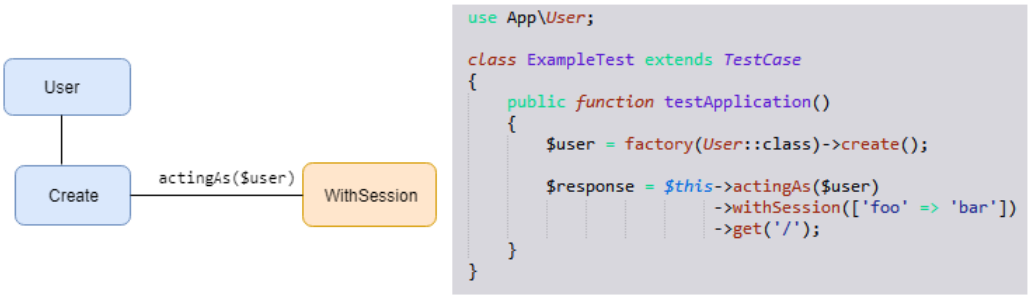


Figura 3. Teste de autenticação de usuário (Fonte: dos autores, 2017)

4.1.3. Testes de Integração

O teste de integração, em suma, verifica se cada uma das funcionalidades está integrada com as funcionalidades que necessitam de outras partes, como a integração do código com o banco de dados e com todas as outras funções que a aplicação irá possuir como requisitos.

A Figura 4 representa um modelo de teste de integração. Neste exemplo utilizam-se dados da aplicação por meio da criação de usuário. A inserção de dados de um possível novo usuário ocorre por meio da classe *User*, que possibilita a conexão com o banco de dados por meio da tabela de dados do usuário. Considerando a proteção dos dados, leva-se em consideração a parametrização dos dados que será detalhada na seção 4.2. Neste caso, em relação à parametrização, faz-se uso de um *array* de dados (*$fillable*) contendo os campos que serão necessários para a criação de um usuário, ou seja, os dados que se deseja testar. Os campos dos dados de teste são armazenados no banco de dados da aplicação. A utilização da biblioteca *DatabaseMigrations* serve para marcar o início e o fim do teste. Com o auxílio da *DatabaseTransactions* os dados armazenados não são salvos, porque a base de dados após o teste retorna ao estado anterior ao mesmo e, assim, o teste pode se repetir. Caso contrário o teste pode vir a falhar, dependendo de suas regras, por exemplo, se ocorrerem eventuais duplicações de dados.

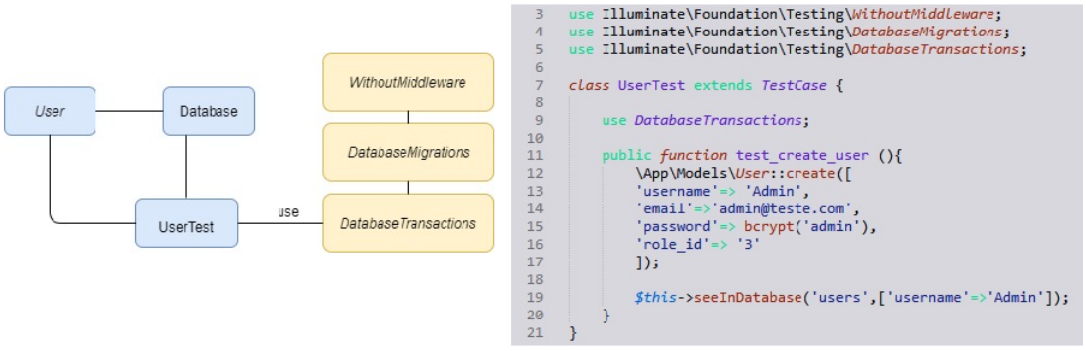
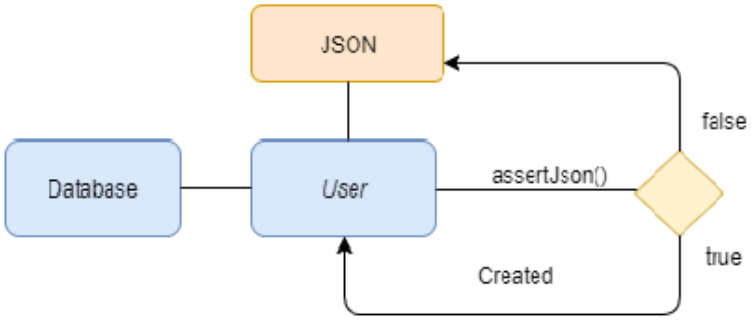


Figura 4. Teste de banco de dados - Inserção de dados de usuário (Fonte: dos autores, 2017)

Após a inserção dos dados de usuário é utilizada a pseudo-variável *this*, para visualizar se o dado especificado foi armazenado na base de dados e foi inserido corretamente. A afirmação, neste caso, verifica na base de dados, tabela *users*, se o usuário 'Admin' foi inserido na tabela. Com as afirmações, as asserções e a parametrização dos dados, podem ser realizados diversos casos de teste, a partir de um caso de teste, alterando os dados das variáveis disponíveis ou modificando o que se espera como resultado do teste por meio das asserções e afirmações.

A Figura 5 apresenta a utilização de métodos auxiliares que o *framework Laravel* oferece, sendo possível realizar os testes com maior agilidade. Um destes métodos auxiliares é o método *post*. Com a utilização da API *JavaScript Object Notation* (JSON) é possível, por meio deste método, e utilizando as asserções (*assertJson)*, analisar se o dado resultante é correspondente ao esperado.




```php
class AssertTest extends TestCase{

    public function testBasicExample(){
        $response = $this->json('POST', '/user', ['username' => 'Admin']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

Figura 5. Diagrama de atividades de teste de APIs JSON - Saídas esperadas (Fonte: dos autores, 2017)

A API JSON especifica como podem ser solicitados recursos para que sejam modificados ou recuperados dados e como um servidor deve responder a essas solicitações, visando minimizar o número de solicitações e a quantidade dos dados cliente/servidor. O formato de troca de dados JSON é simples e eficiente, facilitando a leitura, a escrita e a análise dos dados durante as transações. O processo de transação dos dados ocorre em cinco etapas: (1) servidor analisa os dados que recebe das solicitações; (2) após analisar os dados, eles são deserializados na aplicação; (3) o serviço *web* específico é invocado; (4) os resultados são serializados e transferidos para JSON; e (5) os dados são gravados em uma saída de fluxo de HTTP e enviados para o lado cliente [Peng et al. 2011].

A Figura 6 apresenta um exemplo de um teste de armazenamento de uma possível imagem de perfil de um usuário, de acordo com o modelo de acesso ao banco de dados inicial. O armazenamento ocorre de maneira simulada (armazenamento falso). Nesse processo é utilizado o método *Storage*, que permite a manipulação de arquivos e do método *fake*, que por padrão, elimina arquivos temporários. Além da junção destes métodos, utiliza-se o envio de arquivo através do JSON via *post* para a classe *User*, que possui o campo da imagem de perfil nomeado como avatar. O método *fake* fará com que os arquivos temporários não permaneçam armazenados no banco de dados da aplicação. O teste é analisado por meio de uma asserção, em que, se for possível adicionar a nova imagem ao perfil do usuário testado, o teste será verdadeiro e será simulado o *upload* do arquivo. Caso não seja possível o teste retornará falso.

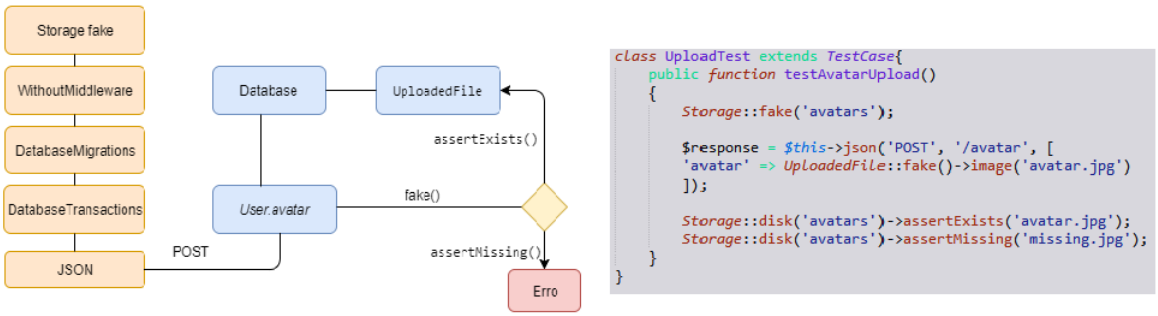


Figura 6. Teste de upload (simulação) (Fonte: dos autores, 2017)

4.2. Tipos especiais de Teste no *Laravel*

Esta seção apresenta as funcionalidades especiais de teste disponíveis no *framework Laravel*, a parametrização dos dados, o modo de conexão com o banco de dados e a aplicação da técnica *pairwise* para implementação de testes negativos.

4.2.1. Parametrização de Testes e Banco de Dados

A utilização de testes parametrizados, em testes automatizados, é um fator relevante no *framework Laravel* quando se deseja passar dados (parâmetros) diferentes em um único cenário de teste. Considerando diversos cenários de desenvolvimento e testes, é necessário um tratamento desses dados, para que sejam testados de maneira adequada, principalmente quando se utiliza a automação. Caso não sejam parametrizados, podem ser necessárias alterações no código, durante a realização dos testes, fazendo com que o tempo de teste seja prolongado.

A parametrização de dados possibilita que, com a modificação de um dado de uma variável de teste, sejam criados novos cenários de teste. Além disso, o mesmo teste pode ser executado diversas vezes, alterando somente os valores das variáveis. Existem diversas maneiras de parametrizar dados de teste em diferentes linguagens de programação. O *Laravel* facilita esse processo disponibilizando a conexão direta com o banco de dados, e a disponibilidade de definir quais campos de uma tabela do banco de dados deseja-se

testar. No exemplo da Figura 7, temos a parametrização dos dados de teste de usuário, que foi apresentada no teste de integração da Figura 4.

```
 */
protected $table = 'users';
protected $fillable = ['username', 'email', 'password', 'role_id'];
```

Figura 7. Parametrização em dados de teste de usuário (Fonte: dos autores, 2017)

A variável *fillable* define e especifica quais os campos da tabela *users* serão disponibilizados e podem ser inseridos pelo usuário (durante o teste). A parametrização de dados é fundamental durante os testes, principalmente quando é utilizado um volume maior de dados.

No *framework Laravel* é disponibilizada uma conexão direta com o banco de dados, sendo possível utilizá-los sem a necessidade de um tratamento prévio. O *framework Laravel* utiliza a arquitetura MVC, onde a conexão com o banco de dados ocorre por meio de uma implementação do *ActiveRecord* disponibilizado através do *Eloquent Object-Relational Mapping* (ORM), que provê o desenvolvimento sob o paradigma de orientação a objetos e bancos de dados relacionais. Com a utilização do *Eloquent* ORM, a manipulação (inserção, atualização, busca e exclusão de registros) é facilitada, de modo que cada tabela do banco de dados possua um modelo correspondente. Esse modelo é responsável pela interação com o banco de dados, facilitando a comunicação com a aplicação e, também, a realização dos testes.

Os modelos são considerados classes, e cada modelo possui uma tabela correspondente no banco de dados, com o mesmo nome e, por padrão, é sempre tratada no plural. Conforme os exemplos de testes apresentados, a classe/modelo será *User* e espera-se encontrar a tabela no banco de dados *users*, contendo informações de usuários. A conexão da *Model* com o banco de dados é realizada por meio da variavel *$table, protected $table='users';*. Quando são realizados os testes, deve-se especificar os campos da tabela que serão utilizados no teste para que não ocorram falhas por meio da variável *$fillable, protected $fillable = ['username', 'email', 'password'];*. Outro fator relevante, é a possibilidade de definir qual a conexão com o banco de dados será utilizada pela *Model*, possibilitando a definição de *models* para diversos bancos de dados simultaneamente [Gabardo 2017].

Além disso, é possível utilizar bibliotecas auxiliares, como por exemplo, a biblioteca *DatabaseMigrations* que marca o inicio e o fim do teste, e a biblioteca *DatabaseTransactions* que, automaticamente, ao realizar uma alteração no banco de dados, após o teste, desfaz as alterações, retornando o banco de dados ao estado anterior ao teste.

O *Laravel* disponibiliza asserções para a realização de seus testes. Em testes de banco de dados são disponibilizadas as seguintes asserções:

- *assertDatabaseHas()*: serve para afirmar que determinada tabela contém os dados fornecidos no banco de dados;
- *assertDatabaseMissing()*: afirma que a tabela no banco de dados não contém os dados fornecidos; e

- *assertSoftDeleted()*: afirma que o registro fornecido foi suprimido.

Com a utilização das asserções e funcionalidades fornecidas pelo *framework Laravel*, é possível realizar diversos tipos de testes e integrar processos de teste de diferentes formas, para garantir que a aplicação está integrada de maneira correta e possua uma execução eficiente de acordo com as regras da aplicação.

Um exemplo que apresenta a conexão citada com o banco de dados pode ser visto na Figura 4, e a conexão com o banco de dados, deste mesmo teste, na Figura 7. Pode-se analisar que o teste da Figura 4, linha 19, não utiliza asserções para verificar se o dado inserido existe no banco de dados. A afirmação citada na linha 19 poderia ser realizada de outra forma, por exemplo utilizando uma asserção *$this->assertDatabaseHas($table, array $data)*, que afirma que a tabela contém o dado fornecido.

4.2.2. Testes Negativos

Além dos exemplos de testes destacados anteriormente (unitários e de integração), com asserções positivas, podemos citar a importância de testes negativos, equivalentes ao teste não fazer aquilo que ele não deve fazer. O *framework Laravel* não possui testes negativos, mas possui maneiras de verificar, por meio de asserções, se um dado que não deve corresponder não corresponde ao dado referido, possuindo um resultado de teste positivo. Uma possibilidade para utilizar casos de teste negativos com o *framework Laravel* é fazer com que as asserções disponíveis não sejam verdadeiras, fazendo com que elas, propositalmente, falhem. Esta etapa exige a criação de casos de teste e, consequentemente, a utilização de técnicas de teste.

Um teste negativo, dependendo da variável que será testada, pode ter diversas possibilidades. Considerando um valor qualquer, se este valor for *booleano* ele tem duas possibilidades de teste, verdadeiro ou falso; sendo um valor válido inteiro, de um a dez, poderíamos ter inúmeras possibilidades de valores inválidos, como a de inserir valores negativos ou maiores que dez, caracteres especiais, etc. É necessário verificar, além das variáveis, o cenário em que estão inseridas para determinar as regras de teste e as opções possíveis para cada variável.

Uma técnica que pode ser utilizada para criar regras para que o teste falhe é a técnica *pairwise*, que é um critério de teste baseado em especificações, exigindo que para cada par de parâmetros de entrada de um sistema, seja realizado ao menos um caso de teste [Tai and Lei 2002]. É possivel definir a técnica com base no pressuposto de que as falhas ocorrem, em sua maioria, por interações de, no máximo, dois fatores. Além de ser utilizada em testes de variáveis comuns, a técnica *pairwise* pode ser eficaz em testes que envolvem inúmeras regras de negócios.

Supondo o cenário de desenvolvimento de *software*, apresentado na Figura 8, com duas váriaveis *booleanas*. Têm-se duas possibilidades na variável 1 (A ou B) e duas possibilidades na variável 2 (A ou B). Ambas são relacionadas conforme requisitos ou regras de negócio e dependentes. Por ser um cenário com variáveis simples, e de opções de entrada limitadas, há 2^2 possibilidades de casos de teste. Neste exemplo é apresentado somente uma possibilidade de caso de teste negativo para cada variável, onde é inserido um valor inválido na variável 1, para que todos os testes possuam saídas negativos. Ou-

tras possibilidades podem ser compostas a partir deste cenário de teste, modificando as variáveis negativas.

Combinação de teste	Variável 1	Variável 2
1	A - Inválido	A
2	A - Inválido	B
3	B - Inválido	A
4	B - Inválido	B

Figura 8. Modelo de combinação de testes utilizando a técnica *pairwise*

Utilizar uma técnica de teste, neste caso, é importante para que se possa criar novos cenários de teste, e testar a maior parte deles, considerando inclusive, as regras de negócio que envolvem a aplicação.

4.3. Testes de Sistema

Por fim, é apresentada uma implementação básica de testes de sistema para que seja possível realizar os principais tipos de teste e garantir qualidade na aplicação desenvolvida. A maneira de execução desse tipo de teste pode ser de forma manual ou automatizada.

Testes de sistema dizem respeito ao teste como um todo, com base em suas especificações. Considerando o contexto de orientação a objetos, requisitos de sistema são derivações de artefatos e é importante que, durante o desenvolvimento, as funcionalidades sejam testadas e equivalentes aos requisitos.

Após a realização de todos os testes apresentados anteriormente, o teste de sistema visa simular a utilização do *software* em seu cenário real de aplicação. O teste de sistema deve ser o mais semelhante possível das configurações do ambiente de utilização para que seja confiável. É necessário conhecer todos os requisitos e definir quais casos de teste serão realizados.

Apesar da simulação do funcionamento do *software* ao ambiente e condições reais de uso, testes de sistema são complexos e difíceis de escrever. Além disso, dependendo do tipo de aplicação podem ser extensos.

No decorrer desta subseção são apresentados dois tipos de teste de sistema: manual e automatizado. Ambos exigem um modelo baseado em teste para definir as regras de negócio da aplicação e o que precisa ser testado para garantir que o *software* esteja de acordo com o esperado pelo usuário final.Sempre que um erro é encontrado em um teste de sistema, deve-se corrigir o erro e testar novamente, por isso, testes automatizados possuem vantagens em relação aos testes manuais, possibilitando maior agilidade no processo de teste e diminuindo retrabalho.

4.3.1. Testes Manuais

Os testes manuais são desenvolvidos, geralmente por meio de metodologias de teste ou testes baseados no modelo *Model Based Testing* (MBT). Essas metodologias abordam artefatos, que surgem com a derivação de casos de teste. Estes artefatos são diagramas de casos de uso, descrições de casos de uso, diagramas de sequência ou colaboração para cada caso de uso, diagramas de classes compostos por domínio de aplicação, e um dicionário de dados que descreve cada classe, método e atributo.

Exemplos de testes que podem ser executados para testar as funcionalidades do sistema são os testes de *Graphical User Interface* (GUI), que visam testar as funcionalidades do sistema por meio da interface gráfica do usuário. É necessário que o sistema atenda às especificações do cliente, de acordo com seus requisitos e regras de negócio e corrija possíveis erros. Para isso, é necessária a comparação de tarefas com possíveis resultados esperados como saída, realizando casos de teste.

Um modelo de teste citado por Yuan [Yuan et al. 2007] é o tempo de execução da GUI como *feedback* para a geração de casos de teste, que é obtido a partir da execução de um conjunto de testes. O *feedback* é utilizado para gerar casos de teste adicionais e testar interações entre os eventos da GUI de diversas maneiras. A partir destes testes gera-se um gráfico de interação de eventos gerados por meio de interação de GUI de duas variáveis. Com a execução de um conjunto de testes, o estado de execução de GUI é gravado e utilizado para obter os eventos de interação semântica para a aplicação.

Com esses eventos obtém-se um gráfico de interação semântica de eventos, capazes de gerenciar casos de teste com interações bidirecionais. Com os casos de teste recém gerados são encontradas falhas adicionais.A detecção de falha é alta porque os eventos são gerados e executados para a combinação de eventos em diferentes ordens de execução. A desvantagem deste mecanismo de *feedback* é o foco em aplicações de GUI. A aplicação e os resultados podem ser diferentes em aplicações com lógicas de negócio e uma GUI simples. Conforme são gerados casos de teste de interações múltiplas, é gerado um número de casos de teste ainda maior [Yuan et al. 2007].

A Figura 9 apresenta um diagrama de sequência que representa o comportamento de diversos objetos e estabelece relações e interações dentro de um contexto. Pode ser utilizado como base para a aplicação de casos de teste, simulando entradas e saídas esperadas por um usuário.

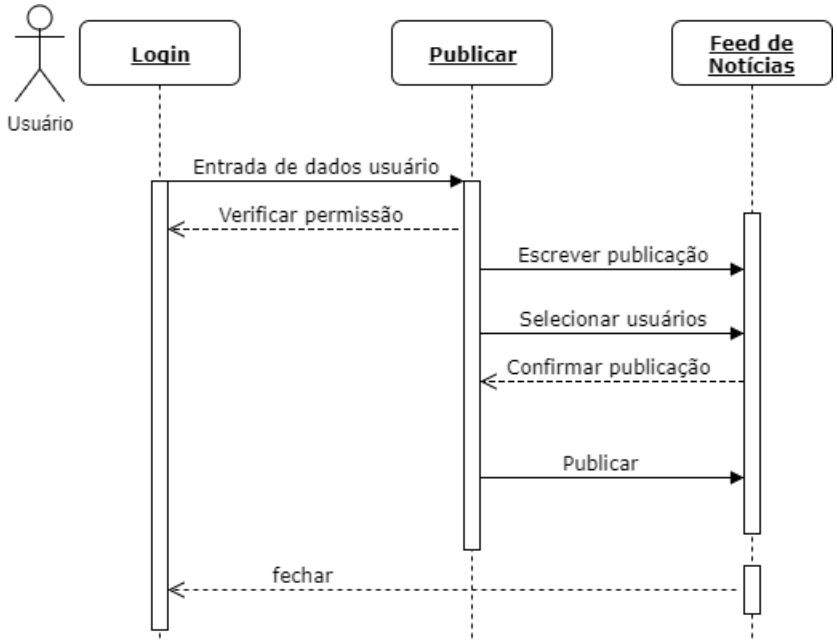


Figura 9. Diagrama de sequência - etapa de especificação de teste manual (Fonte: dos autores, 2017)

4.3.2. Testes Automatizados

A automação de testes faz com que o processo de teste se torne mais eficiente, preciso, rápido e confiável. Testes de sistema automatizados incluem a automatização de tarefas de teste, incluindo as que foram feitas de modo manual, com a utilização de técnicas e ferramentas. O *framework Laravel* apresenta modelos de testes de navegador com a utilização do *Laravel Dusk* e a integração de ferramentas como *Travis CI, CircleCI e Codeship*. Com o auxílio do MBT citado no teste manual e das funcionalidades do *Laravel* é possível automatizar parcialmente as etapas do teste de sistema.

Testes baseados em modelo são considerados uma atividade essencial, tornando-se um método rápido e organizado. É possível fazer um modelo de automatização de testes por meio de técnicas automatizadas de geração e execução de conjuntos de testes e ferramentas. Um modelo de teste que pode ser citado e ulitizado é o modelo de gráfico direcionado do GUI, que é denominado gráfico de interação de eventos, que contém a lógica de negócios da GUI, e a interação semântica de eventos que é usada para identificar o conjunto de eventos que precisam ser testados juntos [Isabella and Retna 2012].

O *Laravel Dusk* facilita testes de integração em páginas *web* de aplicativos habilitados para *JavaScript*, fornecendo caminhos para cliques em *links* e interação com formulários. O *Dusk* cria automaticamente um *Browser* dentro do diretório *tests* do *Laravel* contendo um exemplo de testes. Considerando a instalação automatizada do *Dusk*, por padrão é instalado o *ChromeDriver* para a execução dos testes no navegador. Entretanto, é possível iniciar um servidor *Selenium* e possibilitar a execução dos testes em outros navegadores.

A partir do modelo proposto para a geração de casos de testes MBT, para a execu-

ção destes casos de teste, pode-se utilizar os métodos fornecidos pelo *Laravel Dusk* para automatizar o processo de execução. Os métodos fornecidos pelo *Laravel Dusk* são:

1. *clickLink*: clicar no *link* que contém a informação de texto fornecida;
2. *value*: Interação com os elementos da página, recuperação de texto e atributos;
3. *type*: interação com formulários;
4. *attach*: anexar arquivos em um elemento de entrada;
5. *keys*: fornecer sequências de entrada mais complexas, mantendo teclas modificadoras ativas, como *shift*;
6. *click*: "clicar"em elementos correspondentes ao seletor fornecido no código;
7. *mouseover*: mover o *mouse* sobre elementos que correspondem ao seletor fornecido no código para verificar suas funções;
8. *drag*: arrastar e soltar elementos correspondentes ao seletor;
9. *with*: afirmar que um bloco de texto existe em uma determinada tabela;
10. *pause*: pausar o teste por um número determinado de milissegundos;
11. *waitFor*: pausar a execução do teste até que o elemento correspondente ao seletor seja exibido;
12. *whenAvailable*: esperar por um seletor e interagir com o elemento, também podem ser utilizadas váriaveis deste método, como *waitForText*, *waitForLink*, *waitForLocation* e *waitForReload*; e
13. *waitUntil*: interromper a execução de um teste até que determinada expressão de *JavaScript* seja avaliada.

A Figura 10 demonstra um teste simples de sistema. É realizada a inserção do *e-mail* (linha 16) e é chamado o método *visit*, para ir até a página de *Login* do sistema. Posterior a isso, é realizada a chamada do *e-mail* acima para a inserção dos dados de usuário. O método *press* é utilizado para "clicar"no botão *Login*. A asserção final (linha 23) verifica se foi possível acessar a página inicial (*home*) do sistema. O teste será verdadeiro caso seja possível acessar a página inicial ou virá a falhar caso não seja possível acessá-la.

```
14      public function testBasicExample(){
15          $user = factory(User::class)->create([
16              'email' => 'admin@teste.com',
17          ]);
18          $this->browse(function ($browser) use ($user) {
19              $browser->visit('/login')
20                      ->type('email', $user->email)
21                      ->type('password', 'secret')
22                      ->press('Login')
23                      ->assertPathIs('/home');
24          });
25      }
```

Figura 10. Exemplo de teste visitando a página de Login e acessando a página inicial do usuário (Fonte: dos autores, 2017)

5. Considerações Finais

Neste trabalho foram estudadas as principais técnicas de teste de software disponíveis no *framework Laravel*, visando demonstrá-las por meio de sua utilização, aplicando-as em exemplos simples de aplicações *web*. Observou-se que o *framework Laravel* fornece

diversos caminhos para a execução de testes automatizados, preocupa-se com o desenvolvimento com código limpo e facilita a parametrização dos dados.

Com o estudo dos tipos de teste que o *framework Laravel* fornece, é possível concluir que existem vantagens em se utilizar os tipos de teste fornecidos por este *framework*. Considerando que o desenvolvimento ocorre sob o paradigma de orientação a objetos, aplicando-se técnicas de teste e, envolvendo estes tipos de teste em um plano de teste, pode-se testar o sistema como um todo e automatizar grande parte do processo. Além disso, o *framework Laravel* possui um padrão de desenvolvimento com a utilização da arquitetura MVC, o que facilita o desenvolvimento de *software* e a realização de testes automatizados, visando obter aplicações mais confiáveis.

Outra vantagem que pode ser citada é a integração com as bibliotecas para teste de *software* e a parametrização dos dados com a utilização do *Eloquent ORM* que disponibiliza conexão direta com o banco de dados e, com isso, pode-se ter mais agilidade no processo de teste sem a necessidade de um tratamento prévio dos dados.

Embora existam diversas possibilidades de teste e meios para a automatizar o teste de *software* no *Laravel*, os tipos de teste de *software* apresentados neste trabalho são específicos para o *Laravel*, podendo ser adaptados ambientes de desenvolvimento que utilizam a arquitetura MVC.

Durante o desenvolvimento do trabalho foram encontradas algumas dificuldades em relação à falta de trabalhos relacionados ao teste de *software* no *framework Laravel*, bem como as atualizações constantes de versão e apresentação de novos métodos de teste para a aplicação.

Destacam-se como possíveis trabalhos futuros o desenvolvimento de um plano de teste utilizando os tipos de teste estudados e apresentados a partir da Seção 4 e sua aplicação.

Referências

[Bastos et al. 2012] Bastos, A., Rios, E., Cristalli, R., and Moreira, T. (2012). *Base de Conhecimento em Teste de Software*. Martins Editora Livraria Ltda.

[Bertolino 2007] Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *Future of Software Engineering (FOSE'07)*.

[Gabardo 2017] Gabardo, A. C. (2017). *Laravel para Ninjas*, volume 1. Novatec, São Paulo, Brasil.

[Gouveia 2004] Gouveia, C. C. (2004). Teste de integração para sistemas baseados em componentes. *Simpósio Brasileiro de Engenharia de*.

[Isabella and Retna 2012] Isabella, A. and Retna, E. (2012). Study paper on test case generation for gui based testing. *arXiv preprint arXiv:1202.4527*.

[Karam et al. 2006] Karam, M., Keirouz, W., and Hage, R. (2006). An abstract model for testing mvc and workflow based web applications. In *AICT/ICIW*, page 206. IEEE Computer Society.

[Kersten and Goedicke 2010] Kersten, B. and Goedicke, M. (2010). Browser-based analysis of web framework applications. In Salaün, G., Fu, X., and Hallé, S., editors, *TAV-WEB*, volume 35 of *EPTCS*, pages 51–62.

[Laravel 2017] Laravel (2017). Laravel - the php framework for web artisans.

[Leicht et al. 2017] Leicht, N., Blohm, I., and Leimeister, J. M. (2017). Leveraging the power of the crowd for software testing. *IEEE Software*, 34(2):62–69.

[Maldonado et al. 2004] Maldonado, J. C., Barbosa, E. F., Vincenzi, A. M. R., Delamaro, M. E., Souza, S., and Jino, M. (2004). Introduçao ao teste de software. *São Carlos*, page 23.

[Myers et al. 2011] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

[Neto and Arilo 2006] Neto, D. and Arilo, C. (2006). Uma infra-estrutura computacional para apoiar o planejamento e controle de testes de software.

[Peng et al. 2011] Peng, D., Cao, L., and Xu, W. (2011). Using json for data exchanging in web service applications. *Journal of Computational Information Systems*, 7(16):5883–5890.

[Pressman 2005] Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

[Saunier 2014] Saunier, R. (2014). *Getting Started with Laravel 4*. Packt Publishing Ltd.

[Silva 2008] Silva, D. A. G. (2008). Evolunit: geração e evolução de testes de unidade em java utilizando algoritmos genéticos.

[Tai and Lei 2002] Tai, K.-C. and Lei, Y. (2002). A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111.

[Teixeira et al. 2014] Teixeira, F. A., Pereira, F., Vieira, G., Marcondes, P., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. (2014). Siot – defendendo a internet das coisas contra exploits. *Anais do 32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos – SBRC 2014*.

[Verma 2014] Verma, A. (2014). MVC Architecture: A Comparative Study Between Ruby on Rails and Laravel. *Indian Journal of Computer Science and Engineering (IJCSE)*.

[Way 2013] Way, J. (2013). *Base de Conhecimento em Teste de Software*. Lean Publishing.

[Wong et al. 1995] Wong, W. E., Mathur, A. P., and Maldonado, J. C. (1995). Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness.

[Yuan et al. 2007] Yuan, X., Cohen, M., and Memon, A. M. (2007). Covering array sampling of input event sequences for automated gui testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 405–408. ACM.