Curso de PhpUnit com Laravel

Versão 1.0

Prof. Glauco Todesco

Aula00 – Ambiente

1. Laravel: 8.1.0
2. Php: 7.4.9
3. Composer: 1.10.10
4. MySQL:
5. Visual Studio Code
6. XAMMP:
7. PhpUnit: 9.4.1

Links:

PhpUnit -> <https://phpunit.de/>

<https://laravel.com/docs/8.x/testing>

Aula01 – Criando um projeto

1. Na linha de comando crie um projeto Laravel:

**create-project laravel/laravel cursophpunit**

1. Acesse a pasta:

**cd cursophpunit**

1. Abra o projeto no VSCode:

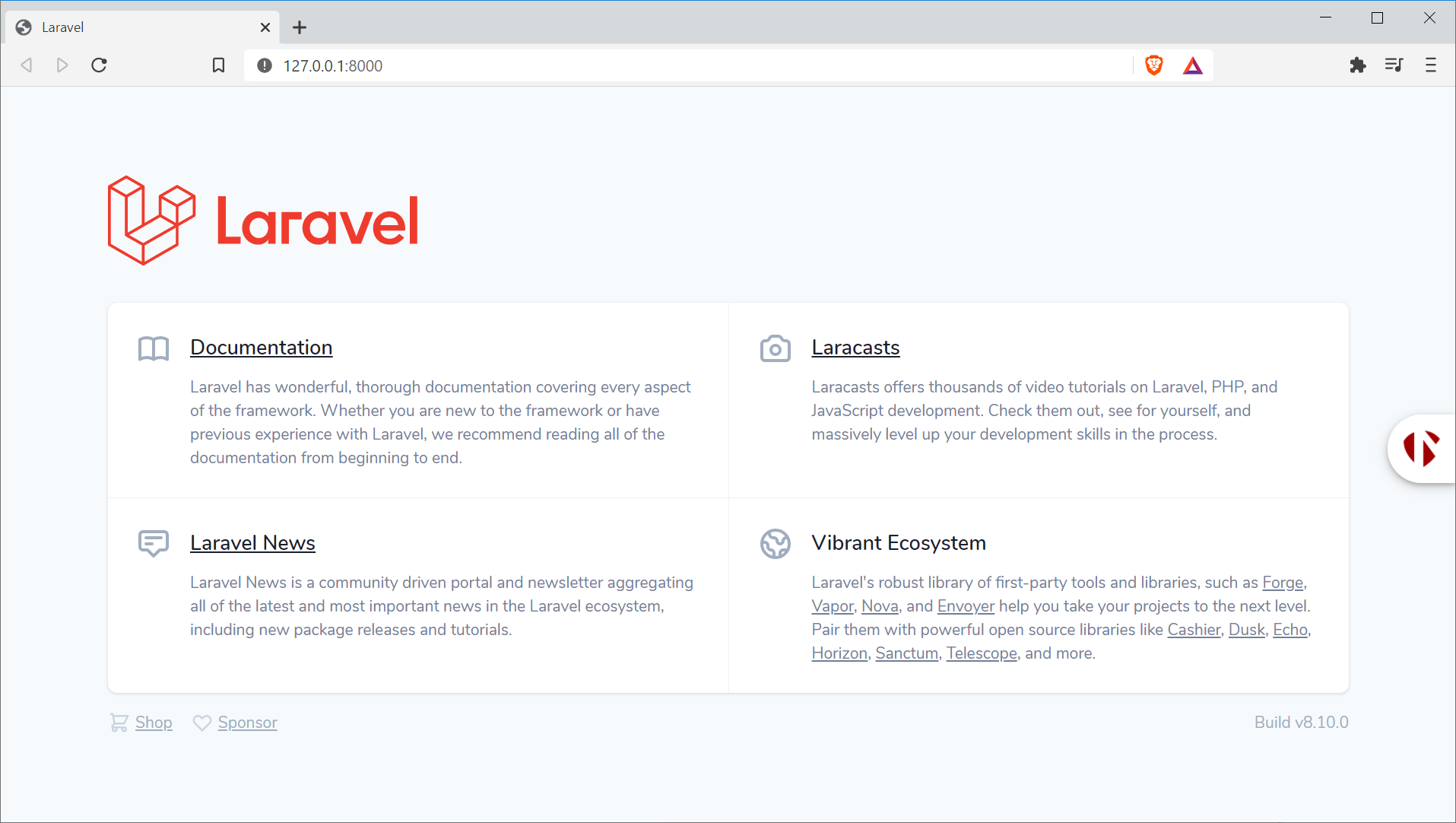
**code .**

1. Abra um terminal no VSCode e execute a aplicação com o servidor do artisan:

**php artisan serve**

1. Abra a aplicação pelo navegador:

**http://127.0.0.1:8000/**



Aula02 – Analisando a estrutura do projeto

1. O Laravel usa o PHPUnit para realizar os testes automatizados.
2. A configuração do PHPUnit é realizada através do arquivo **phpunit.xml** localizado na raiz do projeto.

<?xml version="1.0" encoding="UTF-8"?>

<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:noNamespaceSchemaLocation="./vendor/phpunit/phpunit/phpunit.xsd"

         bootstrap="vendor/autoload.php"

         colors="true"

>

    <testsuites>

        <testsuite name="Unit">

            <directory suffix="Test.php">./tests/Unit</directory>

        </testsuite>

        <testsuite name="Feature">

            <directory suffix="Test.php">./tests/Feature</directory>

        </testsuite>

    </testsuites>

    <coverage processUncoveredFiles="true">

        <include>

            <directory suffix=".php">./app</directory>

        </include>

    </coverage>

    <php>

        <server name="APP\_ENV" value="testing"/>

        <server name="BCRYPT\_ROUNDS" value="4"/>

        <server name="CACHE\_DRIVER" value="array"/>

        <!-- <server name="DB\_CONNECTION" value="sqlite"/> -->

        <!-- <server name="DB\_DATABASE" value=":memory:"/> -->

        <server name="MAIL\_MAILER" value="array"/>

        <server name="QUEUE\_CONNECTION" value="sync"/>

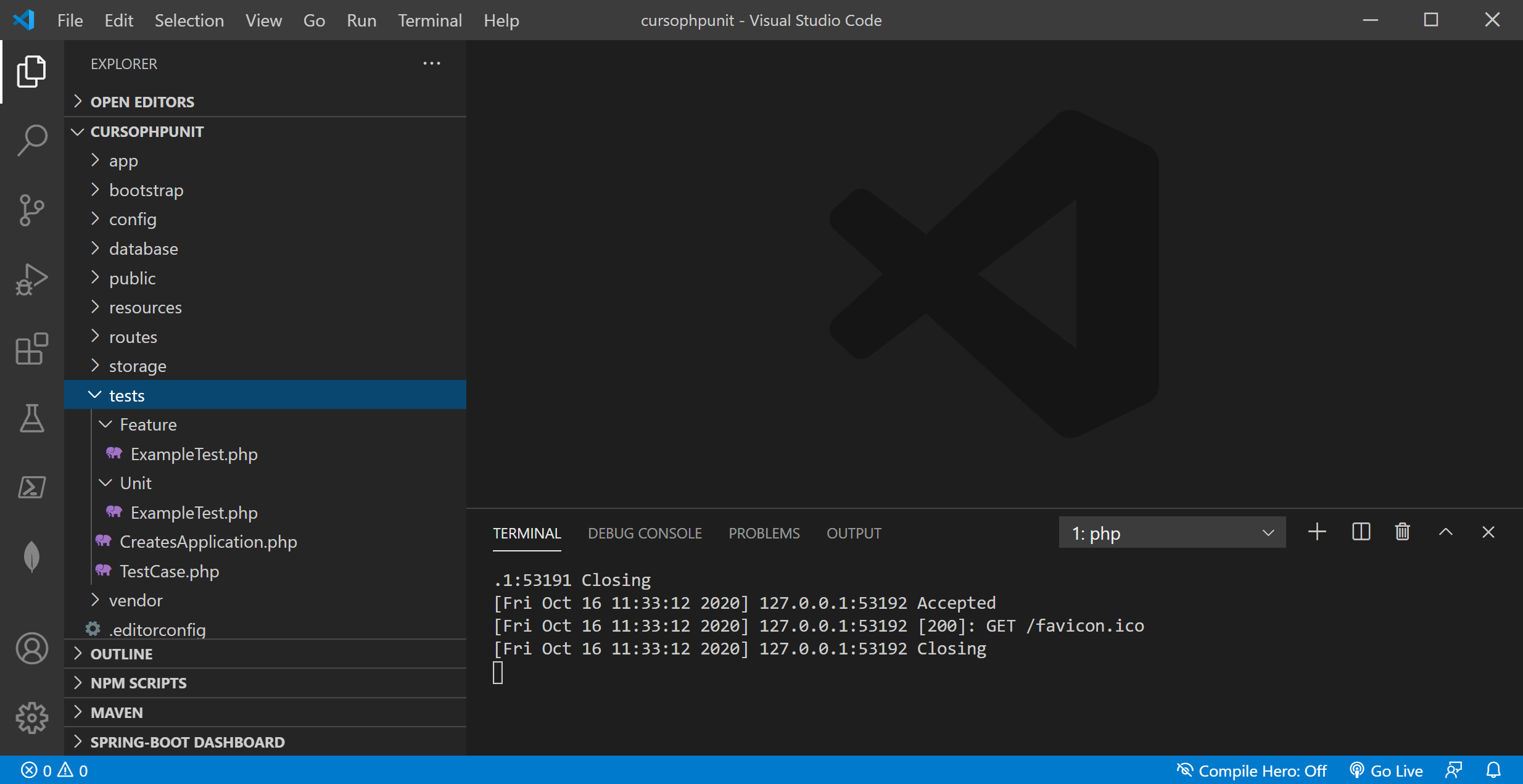
        <server name="SESSION\_DRIVER" value="array"/>

        <server name="TELESCOPE\_ENABLED" value="false"/>

    </php>

</phpunit>

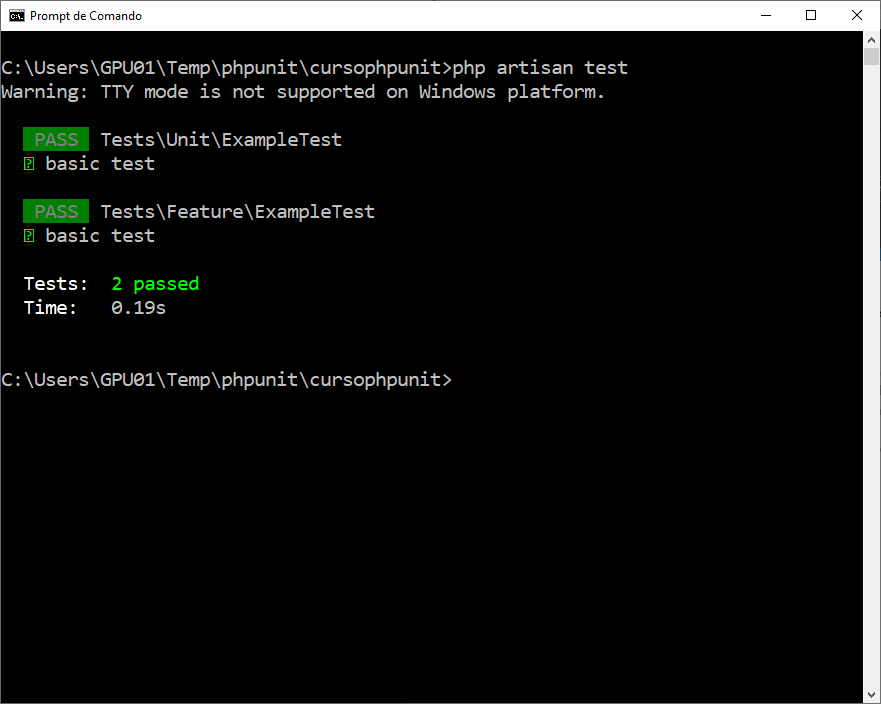
1. Todos os testes deverão ser colocados dentro da pasta **tests** do projeto:



1. O projeto já vem com alguns exemplos criados. Abra e veja os exemplos.
2. A pasta de testes contém dois diretórios: F**eature** e **Unit**. Testes localizados dentro da pasta **Unit** são testes de unidades. Uma unidade de teste é o menor teste que podemos fazer dentro de um projeto, geralmente testando um método de uma classe. Testes localizados dentro da pasta **Feature** são testes mais complexos envolvendo a interação de diversos objetos, requisições HTTP, arquivos JSON e acesso ao banco de dados.
3. Para executar os testes de exemplos, execute o seguinte comando:

**vendor\bin\phpunit** ou **php artisan test**

1. Uma saída parecida com essa será apresentada, no qual os dois testes existentes passaram



1. Vamos fazer uma alteração no teste Feature, no qual o status esperado seja o 404 (Not Found).

<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Tests\TestCase;

class ExampleTest extends TestCase

{

    /\*\*

     \* A basic test example.

     \*

     \* @return void

     \*/

    public function testBasicTest()

    {

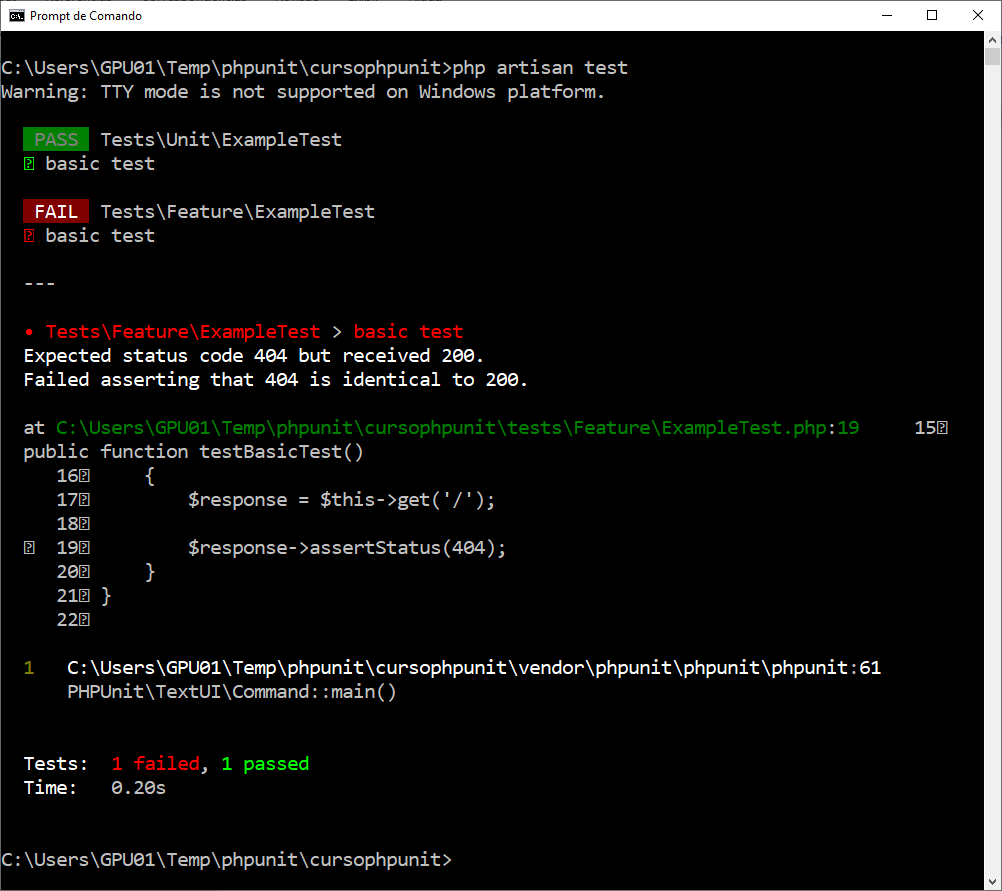
        $response = $this->get('/');

        $response->assertStatus(404);

    }

}

1. Execute os testes novamente e veja o resultado. Agora um teste falhou e o outro passou.



1. Criando um teste de unidade:

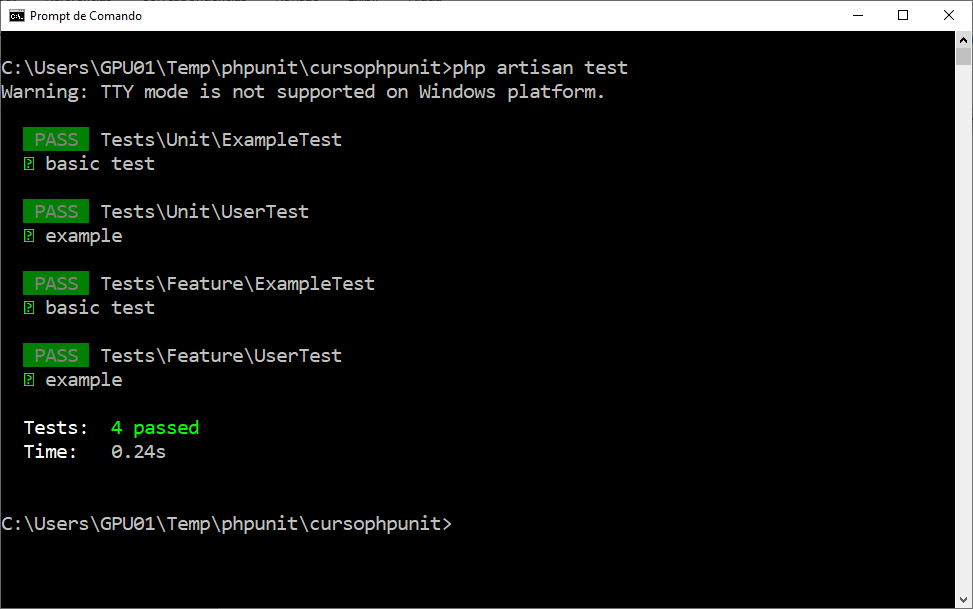
**php artisan make:test UserTest --unit**

1. Criando um teste de feature:

**php artisan make:test UserTest**

1. Temos agora 4 testes. Execute os testes e veja os resultados:

**php artisan test**



Aula 03 – Configuração das demais dependências do projeto.

1. Vamos configurar o projeto com as demais dependências:
2. Crie a base de dados no MySQL blogdb. Configure a base de dados no **.env**
3. Execute o migration para criar as tabelas.
4. Adicione o front-end com bootstrap e a opção de autenticação:
   1. **composer require laravel/ui**
   2. **php artisan ui bootstrap --auth**
   3. **npm install && npm run dev**
5. Adicione o forms:
   1. **composer require laravelcollective/html**
6. Adicione a tradução em português do laravel
7. Instale o dusk para realizar testes de nagevação
   1. **composer require --dev laravel/dusk**
   2. **php artisan dusk:install**
8. Configure o .**env** para que o dusk funcione corretamente.
   1. **APP\_URL=http://localhost:8000**

Aula 04 – Testando Rotas Blog App

1. As seguintes rotas serão testadas:
   1. /services
   2. /projects
   3. /about
   4. /home (somente logado) -> redireciona para /login
   5. /register
   6. /login
   7. /password/reset
2. Vamos criar um teste de Feature para as rotas acima.

**php artisan make:test RouteLogoutTest**

1. Apague o teste de exemplo:

    public function testExample()

    {

        $response = $this->get('/');

        $response->assertStatus(200);

    }

1. Crie a função teste seguinte o padrão abaixo com no máximo 100 caracteres:

**ActionVerb + WhoOrWhatToDO + ExpectedBehavior**

1. Use a anotação @teste para não escrever test em todas as funções

  /\*\* @test \*/

    public function  checkRoutesForAllLogoutUserExist()

    {

    }

1. Para testar se todas as rotas existem, vamos usar o **assertStatus()**

        $this->get('/')->assertStatus(200);

        $this->get('/about')->assertStatus(200);

        $this->get('/projects')->assertStatus(200);

        $this->get('/services')->assertStatus(200);

        $this->get('/login')->assertStatus(200);

        $this->get('/register')->assertStatus(200);

        $this->get('/password/reset')->assertStatus(200);

1. Já a rota **/home** devemos testar se ela redireciona para a rota **/login** usando **assertRedirect()**

 $this->get('/home')->assertRedirect('/login');

1. Execute todos os testes:

**php artisan test**

1. Exercício: Implementes todas as rotas até passar todos os testes

Aula 05 - Testando o modelo User para prevenir ataques (Teste de Unidade)

Link: <https://laravel.com/docs/8.x/eloquent#mass-assignment>

1. Abra a classe **User.php**
2. O trecho de código abaixo define que os campos que podem ser definidos (preenchidos) pelos usuários:

protected $fillable = [

        'name',

        'email',

        'password',

    ];

1. Vamos criar um teste para verificar isso:

**php artisan make:test UserTest --unit**

1. Crie a função abaixo para testarmos se a estrutura é a correta:

    /\*\* @test \*/

    public function checkIfUserColumnsIsCorrect()

    {

        $user = new User();

        $expected = ['name','email','password'];

        $results = array\_diff($expected,$user->getFillable());

        $this->assertEquals(0,count($results));

    }

1. Usamos o assertEquals para fazer esse teste.
2. Execute o teste acima, como essa estrutura já está definida o esperado é passar no teste de primeira.

**php artisan test**

1. Para executar somente um teste use a opção **--filter + *nome\_do\_metodo***

**php artisan test --filter checkIfUserColumnsIsCorrect**

Aula 06 – Criando Testes de Navegação (Dusk)

Link: <https://laravel.com/docs/8.x/dusk>

1. O Dusk permite construir testes de navegação. Ele abre o Google Chrome (sem interface gráfica) e permitindo testar a navegação da aplicação.
2. Construa uma classe para testar as rotas:

**php artisan dusk:make RotasTest**

1. Apague o teste de exemplo e crie um teste para ver conteúdo da rota “/” (Raiz). Observe que a função recebe como parâmetro **$browser**

    /\*\* @test \*/

     public function checkIfRootSiteHasIndexContent()

     {

         $this->browse(function (Browser $browser) {

         });

     }

1. Usando a função **visit**, vamos pedir para navegar até o conteúdo “/” e usando o **assertSee** vamos verificar se o **view** de destino existe um conteúdo esperado:

    /\*\* @test \*/

     public function checkIfRootSiteHasIndexContent()

     {

         $this->browse(function (Browser $browser) {

             $browser->visit('/')

                     ->assertSee('Index!');

         });

     }

1. Execute o teste acima:

**php artisan dusk --filter checkIfRootSiteHasIndexContent**

1. Exercício: Caso tenha erros nos testes, faça a correção do **view** para a rota raiz.
2. Construa outro teste para as demais rotas:

  /\*\* @test \*/

    public function checkIfRotesHasContent()

    {

        $this->browse(function (Browser $browser) {

            $browser->visit('/projects')->assertSee('Projects!')

                    ->visit('/about')->assertSee('About!')

                    ->visit('/services')->assertSee('Services!');

        });

    }

1. Exercício: Caso tenha erros nos testes, faça a correção do **view** para as rotas acima.

Aula 07 – Criando Testes de Formulário (Dusk)

1. O Dusk permite também testar formulários.
2. Crie um teste com o dusk para testar o Registro e Login de um usuário:

**php artisan dusk:make RegisterLoginTest**

1. Apague o teste de exemplo e crie o seguinte teste:

public function checkIfLoginIsWorking()

{

        $this->browse(function (Browser $browser) {

        }

}

1. Dentro da função de teste acima, vamos testar se é possível fazer login de um usuário.
2. Primeiro é necessário incluir o User no arquivo de teste:

use App\Models\User;

1. Para fazer o login com sucesso, é necessário ter um usuário cadastrado. Vamos fazer uma busca de um usuário por email, se não existir esse usuário vamos remover e criá-lo novamente, pois precisamos conhecer a senha para fazer o login com sucesso.

   /\*\* @test \*/

    public function checkIfLoginIsWorking()

    {

        $this->browse(function (Browser $browser) {

            $user = User::where('email','teste1@teste.com')->first();

            if($user != null)

                $user->delete();

            $user = User::factory()->create([

                'email'     => 'teste1@teste.com',

                'name'      => 'Teste'

            ]);

        });

    }

1. A última etapa é fazer o login e verificar se redirecionou para o path “/home”. Para “digitar” dados no formulário usamos a função **type**. Para “clicar”no botão, buscamos usamos a função **press**. Para verificar o sucesso, testamos com **assertPathIs** que o caminho espero é “/home“ após o login. Observe que na última linha é realizado o logout do usuário.

  /\*\* @test \*/

    public function checkIfLoginIsWorking()

    {

        $this->browse(function (Browser $browser) {

            $user = User::where('email','teste1@teste.com')->first();

            if($user != null)

                $user->delete();

            $user = User::factory()->create([

                'email'     => 'teste1@teste.com',

                'name'      => 'Teste'

            ]);

            $browser->visit('/login')

                    ->type('email', $user->email)

                    ->type('password','password')

                    ->press('Login')

                    ->assertPathIs('/home')

                    ->logout();

        });

    }

1. Execute o teste acima:

**php artisan dusk --filter check\_ifLogin\_isWorking**

1. Vamos agora testar o formulário de registro. Crie a função de teste abaixo:

    /\*\* @test \*/

    public function checkIfRegisterIsWorking()

    {

        $this->browse(function (Browser $browser) {

        });

    }

1. O processo é parecido com o login. Vamos verificar se existe um usuário pelo email e remover caso exista, pois não é possível ter dois usuários com mesmo email. Na sequência preenchemos o formulário de registro, pressionamos o botão “Register” testamos se a rota foi redirecionada para o “/home” (sucesso). O último passo foi fazer o logout do usuário.

/\*\* @test \*/

    public function checkIfRegisterIsWorking()

    {

        $this->browse(function (Browser $browser) {

             $user = User::where('email','teste2@teste.com')->first();

            if ($user) {

                $user->forceDelete();

            }

            $browser->visit('/register')

                    ->type('name','Teste2')

                    ->type('email','teste2@teste.com')

                    ->type('password','password')

                    ->type('password\_confirmation','password')

                    ->press('Register')

                    ->assertPathIs('/home')

                    ->logout();

        });

    }

1. Execute o teste acima:

**php artisan dusk --filter checkIfRegisterIsWorking**

Aula 08 – Criando um comando no Artisan

1. Na linha de comando execute:

**php artisan make:command RunTestsCommand**

1. No projeto em “**app/Console/Commands**” foi criado o arquivo **RunTestsCommand**
2. Abra o arquivo **RunTestsCommand** e defina o nome do comando:

  protected $signature = 'tests';

1. Defina a descrição do comando:

 protected $description = 'Execute all tests';

1. Defina o que será executado:

public function handle()

    {

        Artisan::call('test');

        $this->info(Artisan::output());

        Artisan::call('dusk');

        $this->info(Artisan::output());

        return 0;

    }

1. Veja agora todos os comandos:

**php artisan**

1. Execute o comando criado:

**php artisan tests**

Aula 09 – GitHub

1. Para executar as etapas abaixo é necessário ter o seu projeto no GitHub
2. Entre no repositório, e selecione o menu Action
3. Escolha a opção com Laravel:



1. Será criado um arquivo **laravel.yml** com as definições do workflow. Faça o commit e click novamente em Action para ver o resultado da execução do workflow
2. Alterando o arquivo laravel.yml para usar o MySQL e o Dush.
   1. Para configurar o MySQL com um serviço:

services:

      mysql:

        image: mysql:5.7

        env:

          MYSQL\_ALLOW\_EMPTY\_PASSWORD: yes

          MYSQL\_DATABASE: blogdb

        ports:

          - 33306:3306

        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-timeout=5s --health-retries=3

* 1. Atualização do Dusk:

    - name: Update Dusk

      run: php artisan dusk:chrome-driver

* 1. Execução dos testes:

  - name: Execute tests (Unit and Feature tests) via PHPUnit/Dusk

      env:

       DB\_PORT: ${{ job.services.mysql.ports[3306] }}

       DB\_USERNAME: root

       APP\_URL: http://localhost:8000

      run:

        php artisan migrate & php artisan test & php artisan serve & php artisan dusk

1. Versão final do arquivo laravel.yml

name: Laravel

on:

  push:

    branches: [ main ]

  pull\_request:

    branches: [ main ]

jobs:

  laravel-tests:

    runs-on: ubuntu-latest

    services:

      mysql:

        image: mysql:5.7

        env:

          MYSQL\_ALLOW\_EMPTY\_PASSWORD: yes

          MYSQL\_DATABASE: blogdb

        ports:

          - 33306:3306

        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-timeout=5s --health-retries=3

    steps:

    - uses: actions/checkout@v2

    - name: Copy .env

      run: php -r "file\_exists('.env') || copy('.env.example', '.env');"\

    - name: Install Dependencies

      run: composer install -q --no-ansi --no-interaction --no-scripts --no-progress --prefer-dist

    - name: Generate key

      run: php artisan key:generate

    - name: Directory Permissions

      run: chmod -R 777 storage bootstrap/cache

    - name: Update Dusk

      run: php artisan dusk:chrome-driver

    - name: Execute tests (Unit and Feature tests) via PHPUnit/Dusk

      env:

       DB\_PORT: ${{ job.services.mysql.ports[3306] }}

       DB\_USERNAME: root

       APP\_URL: http://localhost:8000

      run:

        php artisan migrate & php artisan test & php artisan serve & php artisan dusk

1. Faça um commit e veja os resultados dos testes.
2. Todas as vezes que o repositório for atualizado, os testes serão realizados!

Aula 10 – Php Code Sniffer com GitHub Actions

1. Existe vários analisadores de código que verificam a conformidade do código fonte em relação a um padrão. O padrão usado pelo Laravel é o PSR-2. Veja a referência no link abaixo:

<https://www.php-fig.org/psr/psr-2/>

1. Vamos usar o seguinte projeto para fazer isso:

<https://github.com/squizlabs/PHP_CodeSniffer>

1. A instalação pode ser feita via Composer:

**composer require --dev squizlabs/php\_codesniffer**

1. Já é possível rodar a ferramenta pela linha de comando abaixo, no qual vai analisar todo o projeto, porém vários erros serão apresentados pois não está sendo usado o padrão do Laravel PSR-2, e sim um outro padrão.

**vendor\bin\phpcs**

1. Precisamos criar um arquivo de configuração do codesniffer **phpcs.xml**. Crie esse arquivo na raiz do projeto com o seguinte conteúdo:

<?xml version="1.0"?>

<ruleset name="PHP\_CodeSniffer">

<description>The coding standard for our project.</description>

<rule ref="PSR2"/>

<file>app</file>

<file>bootstrap</file>

<file>config</file>

<file>database</file>

<file>resources</file>

<file>routes</file>

<file>tests</file>

<exclude-pattern>bootstrap/cache/\*</exclude-pattern>

<exclude-pattern>bootstrap/autoload.php</exclude-pattern>

<exclude-pattern>\*/migrations/\*</exclude-pattern>

<exclude-pattern>\*/seeds/\*</exclude-pattern>

<exclude-pattern>\*.blade.php</exclude-pattern>

<exclude-pattern>\*.js</exclude-pattern>

<!-- Show progression -->

<arg value="p"/>

</ruleset>

1. Rode mais uma vez a ferramenta e veja os erros atuais do seu código:

**.\vendor\bin\phpcs**

1. Alguns erros podem ser corrigidos automaticamente usando o seguinte comando:

**.\vendor\bin\phpcbf**

1. Para incorporar o CodeSniffer no Action do GitHub, acrescente a seguinte linha no arquivo **laravel.yml**, logo após o passo do **composer install**

- name: check PSR2 with CodeSniffer

     run: ./vendor/bin/phpcs

1. Faça o commit e push do projeto e veja o resultado no Actions do GitHub do seu projeto.

Aula 11 – Deployer

<https://deployer.org/>

1. Instale o deployer através do composer:

composer global require deployer/deployer

1. Veja se o comando deployer está funcionando executando o comando **dep** na linha de comando.

C:\Users\GPU01\Temp\phpunit\cursophpunit>dep

Deployer v6.8.0

Usage:

command [options] [arguments]

Options:

-h, --help Display this help message

-q, --quiet Do not output any message

-V, --version Display this application version

--ansi Force ANSI output

--no-ansi Disable ANSI output

-n, --no-interaction Do not ask any interactive question

-f, --file[=FILE] Specify Deployer file

-v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:

autocomplete Install command line autocompletion capabilities

help Displays help for a command

init Initialize deployer in your project

list Lists commands

run Run any arbitrary command on hosts

ssh Connect to host through ssh

debug

debug:task Display the task-tree for a given task

1. Na pasta de projeto vamos digitar o comando para configurar no dep no projeto:

**dep init -t Laravel**

1. Após executar o comando acima foi criado o arquivo deploy.php na raiz do projeto. Vamos configurar esse arquivo.
2. Primeiro precisamos indicar o caminho do repositório do nosso projeto, altere a opção abaixo para o seu repositório:

// Project repository

set('repository', 'https://github.com/glaucotodesco/curso\_phpunit\_laravel.git');

1. Próxima etapa é definir o host do servidor, altere a linha abaixo, definindo o usuário@host. Defina também o diretório destino no servidor

// Hosts

host('ubuntu@54.160.121.154')

    ->set('deploy\_path', '/var/www/html');

<https://github.com/atymic/deployer-php-action>

<https://atymic.dev/blog/github-actions-laravel-ci-cd/>

<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html>

* + - 1. Acesse 1 aws ec2
      2. Crie uma maquina free ubuntu 20
      3. Logue
      4. Atualize: sudo apt-get upgrade

<https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04-pt>

<https://youtu.be/G5Nk4VykcUw>

<https://semaphoreci.com/blog/7-continuous-integration-tools-for-php-laravel>