Laravel Dusk

Introduction

Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your machine. Instead, Dusk uses a standalone ChromeDriver installation. However, you are free to utilize any other Selenium compatible driver you wish.

Installation

To get started, you should add the `laravel/dusk` Composer dependency to your project:

```
composer require --dev laravel/dusk
```

> ! If you are manually registering Dusk's service provider, you should **never** register it in your production environment, as doing so could lead to arbitrary users being able to authenticate with your application.

After installing the Dusk package, run the `dusk:install` Artisan command:

```
php artisan dusk:install
```

A `Browser` directory will be created within your `tests` directory and will contain an example test. Next, set the `APP_URL` environment variable in your `.env` file. This value should match the URL you use to access your application in a browser.

To run your tests, use the `dusk` Artisan command. The `dusk` command accepts any argument that is also accepted by the `phpunit` command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

Managing ChromeDriver Installations

If you would like to install a different version of ChromeDriver than what is included with Laravel Dusk, you may use the `dusk:chrome-driver` command:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 74

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all
```

Dusk requires the `chromedriver` binaries to be executable. If you're having problems running Dusk, you should ensure the binaries are executable using the following command: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Using Other Browsers

By default, Dusk uses Google Chrome and a standalone ChromeDriver installation to run your browser tests. However, you may start your own Selenium server and run your tests against any browser you wish.

To get started, open your `tests/DuskTestCase.php` file, which is the base Dusk test case for your application. Within this file, you can remove the call to the `startChromeDriver` method. This will stop Dusk from automatically starting the ChromeDriver:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

Next, you may modify the `driver` method to connect to the URL and port of your choice. In addition, you may modify the "desired capabilities" that should be passed to the WebDriver:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

Getting Started

Generating Tests

To generate a Dusk test, use the `dusk:make` Artisan command. The generated test will be placed in the `tests/Browser` directory:

```
php artisan dusk:make LoginTest
```

Running Tests

To run your browser tests, use the `dusk` Artisan command:

```
php artisan dusk
```

If you had test failures the last time you ran the `dusk` command, you may save time by re-running the failing tests first using the `dusk:fails` command:

```
php artisan dusk:fails
```

The `dusk` command accepts any argument that is normally accepted by the PHPUnit test runner, allowing you to only run the tests for a given group, etc:

```
php artisan dusk --group=foo
```

Manually Starting ChromeDriver

By default, Dusk will automatically attempt to start ChromeDriver. If this does not work for your particular system, you may manually start ChromeDriver before running the `dusk` command. If you choose to start ChromeDriver manually, you should comment out the following line of your `tests/DuskTestCase.php` file:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

In addition, if you start ChromeDriver on a port other than 9515, you should modify the `driver` method of the same class:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

Environment Handling

To force Dusk to use its own environment file when running tests, create a `.env.dusk.{environment}` file in the root of your project. For example, if you will be initiating the `dusk` command from your `local` environment, you should create a `.env.dusk.local` file.

When running tests, Dusk will back-up your `.env` file and rename your Dusk environment to `.env`. Once the tests have completed, your `.env` file will be restored.

Creating Browsers

To get started, let's write a test that verifies we can log into our application. After generating a test, we can modify it to navigate to the login page, enter some credentials, and click the "Login" button. To create a browser instance, call the `browse` method:

```
<?php

namespace Tests\Browser;

use App\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
```

```
    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $user = factory(User::class)->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                    ->type('email', $user->email)
                    ->type('password', 'password')
                    ->press('Login')
                    ->assertPathIs('/home');
        });
    }
}
```

As you can see in the example above, the `browse` method accepts a callback. A browser instance will automatically be passed to this callback by Dusk and is the main object used to interact with and make assertions against your application.

Creating Multiple Browsers

Sometimes you may need multiple browsers in order to properly carry out a test. For example, multiple browsers may be needed to test a chat screen that interacts with websockets. To create multiple browsers, "ask" for more than one browser in the signature of the callback given to the `browse` method:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home')
          ->waitForText('Message');

    $second->loginAs(User::find(2))
           ->visit('/home')
           ->waitForText('Message')
           ->type('message', 'Hey Taylor')
           ->press('Send');

    $first->waitForText('Hey Taylor')
          ->assertSee('Jeffrey Way');
});
```

Resizing Browser Windows

You may use the `resize` method to adjust the size of the browser window:

```
$browser->resize(1920, 1080);
```

The `maximize` method may be used to maximize the browser window:

```
$browser->maximize();
```

Browser Macros

If you would like to define a custom browser method that you can re-use in a variety of your tests, you may use the `macro` method on the `Browser` class. Typically, you should call this method from a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register the Dusk's browser macros.
     *
     * @return void
     */
    public function boot()
    {
        Browser::macro('scrollToElement', function ($element = null) {
            $this->script("$('html, body').animate({ scrollTop: $('$element').offs

            return $this;
        });
```

```
    }
}
```

The `macro` function accepts a name as its first argument, and a Closure as its second. The macro's Closure will be executed when calling the macro as a method on a `Browser` implementation:

```
$this->browse(function ($browser) use ($user) {
    $browser->visit('/pay')
            ->scrollToElement('#credit-card-details')
            ->assertSee('Enter Credit Card Details');
});
```

Authentication

Often, you will be testing pages that require authentication. You can use Dusk's `loginAs` method in order to avoid interacting with the login screen during every test. The `loginAs` method accepts a user ID or user model instance:

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
          ->visit('/home');
});
```

After using the `loginAs` method, the user session will be maintained for all tests within the file.

Database Migrations

When your test requires migrations, like the authentication example above, you should never use the `RefreshDatabase` trait. The `RefreshDatabase` trait leverages database transactions which will not be applicable across HTTP requests. Instead, use the `DatabaseMigrations` trait:

```
<?php

namespace Tests\Browser;

use App\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
}
```

Interacting With Elements

Dusk Selectors

Choosing good CSS selectors for interacting with elements is one of the hardest parts of writing Dusk tests. Over time, frontend changes can cause CSS selectors like the following to break your tests:

```
// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

Dusk selectors allow you to focus on writing effective tests rather than remembering CSS selectors. To define a selector, add a `dusk` attribute to your HTML element. Then, prefix the selector with `@` to manipulate the attached element within a Dusk test:

```
// HTML...

<button dusk="login-button">Login</button>

// Test...

$browser->click('@login-button');
```

Clicking Links

To click a link, you may use the `clickLink` method on the browser instance. The `clickLink` method will click the link that has the given display text:

```
$browser->clickLink($linkText);
```

This method interacts with jQuery. If jQuery is not available on the page, Dusk will automatically inject it into the page so it is available for the test's duration.

Text, Values, & Attributes

Retrieving & Setting Values

Dusk provides several methods for interacting with the current display text, value, and attributes of elements on the page. For example, to get the "value" of an element that matches a given selector, use the `value` method:

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

Retrieving Text

The `text` method may be used to retrieve the display text of an element that matches the given selector:

```
$text = $browser->text('selector');
```

Retrieving Attributes

Finally, the `attribute` method may be used to retrieve an attribute of an element matching the given selector:

```
$attribute = $browser->attribute('selector', 'value');
```

Using Forms

Typing Values

Dusk provides a variety of methods for interacting with forms and input elements. First, let's take a look at an example of typing text into an input field:

```
$browser->type('email', 'taylor@laravel.com');
```

Note that, although the method accepts one if necessary, we are not required to pass a CSS selector into the `type` method. If a CSS selector is not provided, Dusk will search for an input field with the given `name` attribute. Finally, Dusk will attempt to find a `textarea` with the given `name` attribute.

To append text to a field without clearing its content, you may use the `append` method:

```
$browser->type('tags', 'foo')
        ->append('tags', ', bar, baz');
```

You may clear the value of an input using the `clear` method:

```
$browser->clear('email');
```

Dropdowns

To select a value in a dropdown selection box, you may use the `select` method. Like the `type` method, the `select` method does not require a full CSS selector. When passing a value to the `select` method, you should pass the underlying option value instead of the display text:

```
$browser->select('size', 'Large');
```

You may select a random option by omitting the second parameter:

```
$browser->select('size');
```

Checkboxes

To "check" a checkbox field, you may use the `check` method. Like many other input related methods, a full CSS selector is not required. If an exact selector match can't be found, Dusk will search for a checkbox with a matching `name` attribute:

```
$browser->check('terms');

$browser->uncheck('terms');
```

Radio Buttons

To "select" a radio button option, you may use the `radio` method. Like many other input related methods, a full CSS selector is not required. If an exact selector match can't be found, Dusk will search for a radio with matching `name` and `value` attributes:

```
$browser->radio('version', 'php7');
```

Attaching Files

The `attach` method may be used to attach a file to a `file` input element. Like many other input related methods, a full CSS selector is not required. If an exact selector match can't be found, Dusk will search for a file input with matching `name` attribute:

```
$browser->attach('photo', __DIR__.'/photos/me.png');
```

The attach function requires the `Zip` PHP extension to be installed and enabled on your server.

Using The Keyboard

The `keys` method allows you to provide more complex input sequences to a given element than normally allowed by the `type` method. For example, you may hold modifier keys entering values. In this example, the `shift` key will be held while `taylor` is entered into the element matching the given selector. After `taylor` is typed, `otwell` will be typed without any modifier keys:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

You may even send a "hot key" to the primary CSS selector that contains your application:

```
$browser->keys('.app', ['{command}', 'j']);
```

All modifier keys are wrapped in `{}` characters, and match the constants defined in the `Facebook\WebDriver\WebDriverKeys` class, which can be found on GitHub.

Using The Mouse

Clicking On Elements

The `click` method may be used to "click" on an element matching the given selector:

```
$browser->click('.selector');
```

Mouseover

The `mouseover` method may be used when you need to move the mouse over an element matching the given selector:

```
$browser->mouseover('.selector');
```

Drag & Drop

The `drag` method may be used to drag an element matching the given selector to another element:

```
$browser->drag('.from-selector', '.to-selector');
```

Or, you may drag an element in a single direction:

```
$browser->dragLeft('.selector', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

JavaScript Dialogs

Dusk provides various methods to interact with JavaScript Dialogs:

```
// Wait for a dialog to appear:
$browser->waitForDialog($seconds = null);

// Assert that a dialog has been displayed and that its message matches the given
$browser->assertDialogOpened('value');

// Type the given value in an open JavaScript prompt dialog:
$browser->typeInDialog('Hello World');
```

To close an opened JavaScript Dialog, clicking the OK button:

```
$browser->acceptDialog();
```

To close an opened JavaScript Dialog, clicking the Cancel button (for a confirmation dialog only):

```
$browser->dismissDialog();
```

Scoping Selectors

Sometimes you may wish to perform several operations while scoping all of the operations within a given selector. For example, you may wish to assert that some text exists only within a table and then click a button within that table. You may use the `with` method to accomplish this. All operations performed within the callback given to the `with` method will be scoped to the original selector:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

Waiting For Elements

When testing applications that use JavaScript extensively, it often becomes necessary to "wait" for certain elements or data to be available before proceeding with a test. Dusk makes this a cinch. Using a variety of methods, you may wait for elements to be visible on the page or even wait until a given JavaScript expression evaluates to `true`.

Waiting

If you need to pause the test for a given number of milliseconds, use the `pause` method:

```
$browser->pause(1000);
```

Waiting For Selectors

The `waitFor` method may be used to pause the execution of the test until the element matching the given CSS selector is displayed on the page. By default, this will pause the test for a maximum of five seconds before throwing an exception. If necessary, you may pass a custom timeout threshold as the second argument to the method:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
```

```
$browser->waitFor('.selector', 1);
```

You may also wait until the given selector is missing from the page:

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

Scoping Selectors When Available

Occasionally, you may wish to wait for a given selector and then interact with the element matching the selector. For example, you may wish to wait until a modal window is available and then press the "OK" button within the modal. The `whenAvailable` method may be used in this case. All element operations performed within the given callback will be scoped to the original selector:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
          ->press('OK');
});
```

Waiting For Text

The `waitForText` method may be used to wait until the given text is displayed on the page:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

Waiting For Links

The `waitForLink` method may be used to wait until the given link text is displayed on the page:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Waiting On The Page Location

When making a path assertion such as `$browser->assertPathIs('/home')`, the assertion can fail if `window.location.pathname` is being updated asynchronously. You may use the `waitForLocation` method to wait for the location to be a given value:

```
$browser->waitForLocation('/secret');
```

You may also wait for a named route's location:

```
$browser->waitForRoute($routeName, $parameters);
```

Waiting for Page Reloads

If you need to make assertions after a page has been reloaded, use the `waitForReload` method:

```
$browser->click('.some-action')
        ->waitForReload()
        ->assertSee('something');
```

Waiting On JavaScript Expressions

Sometimes you may wish to pause the execution of a test until a given JavaScript expression evaluates to `true`. You may easily accomplish this using the `waitUntil` method. When passing an expression to this method, you do not need to include the `return` keyword or an ending semi-colon:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitUntil('App.dataLoaded');

$browser->waitUntil('App.data.servers.length > 0');
```

```
// Wait a maximum of one second for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

Waiting On Vue Expressions

The following methods may be used to wait until a given Vue component attribute has a given value:

```
// Wait until the component attribute contains the given value...
$browser->waitUntilVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitUntilVueIsNot('user.name', null, '@user');
```

Waiting With A Callback

Many of the "wait" methods in Dusk rely on the underlying `waitUsing` method. You may use this method directly to wait for a given callback to return `true`. The `waitUsing` method accepts the maximum number of seconds to wait, the interval at which the Closure should be evaluated, the Closure, and an optional failure message:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Making Vue Assertions

Dusk even allows you to make assertions on the state of Vue component data. For example, imagine your application contains the following Vue component:

```
// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',

    data: function () {
        return {
            user: {
              name: 'Taylor'
            }
        };
    }
});
```

You may assert on the state of the Vue component like so:

```
/**
 * A basic Vue test example.
 *
 * @return void
 */
public function testVue()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')
                ->assertVue('user.name', 'Taylor', '@profile-component');
    });
}
```

Available Assertions

Dusk provides a variety of assertions that you may make against your application. All of the available assertions are documented in the list below:

assertTitle	assertCookieMissing	assertSelectHasOptions
assertTitleContains	assertCookieValue	assertSelectMissingOptior
assertUrlIs	assertPlainCookieValue	assertSelectHasOption
assertSchemeIs	assertSee	assertValue
assertSchemeIsNot	assertDontSee	assertVisible
assertHostIs	assertSeeIn	assertPresent
assertHostIsNot	assertDontSeeIn	assertMissing
assertPortIs	assertSourceHas	assertDialogOpened
assertPortIsNot	assertSourceMissing	assertEnabled
assertPathBeginsWith	assertSeeLink	assertDisabled
assertPathIs	assertDontSeeLink	assertButtonEnabled
assertPathIsNot	assertInputValue	assertButtonDisabled
assertRouteIs	assertInputValueIsNot	assertFocused

assertTitle

Assert that the page title matches the given text:

```
$browser->assertTitle($title);
```

assertTitleContains

Assert that the page title contains the given text:

```
$browser->assertTitleContains($title);
```

assertUrlIs

Assert that the current URL (without the query string) matches the given string:

```
$browser->assertUrlIs($url);
```

assertSchemeIs

Assert that the current URL scheme matches the given scheme:

```
$browser->assertSchemeIs($scheme);
```

assertSchemeIsNot

Assert that the current URL scheme does not match the given scheme:

```
$browser->assertSchemeIsNot($scheme);
```

assertHostIs

Assert that the current URL host matches the given host:

```
$browser->assertHostIs($host);
```

assertHostIsNot

Assert that the current URL host does not match the given host:

```
$browser->assertHostIsNot($host);
```

assertPortIs

Assert that the current URL port matches the given port:

```
$browser->assertPortIs($port);
```

assertPortIsNot

Assert that the current URL port does not match the given port:

```
$browser->assertPortIsNot($port);
```

assertPathBeginsWith

Assert that the current URL path begins with the given path:

```
$browser->assertPathBeginsWith($path);
```

assertPathIs

Assert that the current path matches the given path:

```
$browser->assertPathIs('/home');
```

assertPathIsNot

Assert that the current path does not match the given path:

```
$browser->assertPathIsNot('/home');
```

assertRouteIs

Assert that the current URL matches the given named route's URL:

```
$browser->assertRouteIs($name, $parameters);
```

assertQueryStringHas

Assert that the given query string parameter is present:

```
$browser->assertQueryStringHas($name);
```

Assert that the given query string parameter is present and has a given value:

```
$browser->assertQueryStringHas($name, $value);
```

assertQueryStringMissing

Assert that the given query string parameter is missing:

```
$browser->assertQueryStringMissing($name);
```

assertFragmentIs

Assert that the current fragment matches the given fragment:

```
$browser->assertFragmentIs('anchor');
```

assertFragmentBeginsWith

Assert that the current fragment begins with the given fragment:

```
$browser->assertFragmentBeginsWith('anchor');
```

assertFragmentIsNot

Assert that the current fragment does not match the given fragment:

```
$browser->assertFragmentIsNot('anchor');
```

assertHasCookie

Assert that the given cookie is present:

```
$browser->assertHasCookie($name);
```

assertCookieMissing

Assert that the given cookie is not present:

```
$browser->assertCookieMissing($name);
```

assertCookieValue

Assert that a cookie has a given value:

```
$browser->assertCookieValue($name, $value);
```

assertPlainCookieValue

Assert that an unencrypted cookie has a given value:

```
$browser->assertPlainCookieValue($name, $value);
```

assertSee

Assert that the given text is present on the page:

```
$browser->assertSee($text);
```

assertDontSee

Assert that the given text is not present on the page:

```
$browser->assertDontSee($text);
```

assertSeeIn

Assert that the given text is present within the selector:

```
$browser->assertSeeIn($selector, $text);
```

assertDontSeeIn

Assert that the given text is not present within the selector:

```
$browser->assertDontSeeIn($selector, $text);
```

assertSourceHas

Assert that the given source code is present on the page:

```
$browser->assertSourceHas($code);
```

assertSourceMissing

Assert that the given source code is not present on the page:

```
$browser->assertSourceMissing($code);
```

assertSeeLink

Assert that the given link is present on the page:

```
$browser->assertSeeLink($linkText);
```

assertDontSeeLink

Assert that the given link is not present on the page:

```
$browser->assertDontSeeLink($linkText);
```

assertInputValue

Assert that the given input field has the given value:

```
$browser->assertInputValue($field, $value);
```

assertInputValueIsNot

Assert that the given input field does not have the given value:

```
$browser->assertInputValueIsNot($field, $value);
```

assertChecked

Assert that the given checkbox is checked:

```
$browser->assertChecked($field);
```

assertNotChecked

Assert that the given checkbox is not checked:

```
$browser->assertNotChecked($field);
```

assertRadioSelected

Assert that the given radio field is selected:

```
$browser->assertRadioSelected($field, $value);
```

assertRadioNotSelected

Assert that the given radio field is not selected:

```
$browser->assertRadioNotSelected($field, $value);
```

assertSelected

Assert that the given dropdown has the given value selected:

```
$browser->assertSelected($field, $value);
```

assertNotSelected

Assert that the given dropdown does not have the given value selected:

```
$browser->assertNotSelected($field, $value);
```

assertSelectHasOptions

Assert that the given array of values are available to be selected:

```
$browser->assertSelectHasOptions($field, $values);
```

assertSelectMissingOptions

Assert that the given array of values are not available to be selected:

```
$browser->assertSelectMissingOptions($field, $values);
```

assertSelectHasOption

Assert that the given value is available to be selected on the given field:

```
$browser->assertSelectHasOption($field, $value);
```

assertValue

Assert that the element matching the given selector has the given value:

```
$browser->assertValue($selector, $value);
```

assertVisible

Assert that the element matching the given selector is visible:

```
$browser->assertVisible($selector);
```

assertPresent

Assert that the element matching the given selector is present:

```
$browser->assertPresent($selector);
```

assertMissing

Assert that the element matching the given selector is not visible:

```
$browser->assertMissing($selector);
```

assertDialogOpened

Assert that a JavaScript dialog with the given message has been opened:

```
$browser->assertDialogOpened($message);
```

assertEnabled

Assert that the given field is enabled:

```
$browser->assertEnabled($field);
```

assertDisabled

Assert that the given field is disabled:

```
$browser->assertDisabled($field);
```

assertButtonEnabled

Assert that the given button is enabled:

```
$browser->assertButtonEnabled($button);
```

assertButtonDisabled

Assert that the given button is disabled:

```
$browser->assertButtonDisabled($button);
```

assertFocused

Assert that the given field is focused:

```
$browser->assertFocused($field);
```

assertNotFocused

Assert that the given field is not focused:

```
$browser->assertNotFocused($field);
```

assertVue

Assert that a given Vue component data property matches the given value:

```
$browser->assertVue($property, $value, $componentSelector = null);
```

assertVueIsNot

Assert that a given Vue component data property does not match the given value:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

assertVueContains

Assert that a given Vue component data property is an array and contains the given value:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

assertVueDoesNotContain

Assert that a given Vue component data property is an array and does not contain the given value:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);
```

Pages

Sometimes, tests require several complicated actions to be performed in sequence. This can make your tests harder to read and understand. Pages allow you to define expressive actions that may then be performed on a given page using a single method. Pages also allow you to define short-cuts to common selectors for your application or a single page.

Generating Pages

To generate a page object, use the `dusk:page` Artisan command. All page objects will

be placed in the `tests/Browser/Pages` directory:

```
php artisan dusk:page Login
```

Configuring Pages

By default, pages have three methods: `url`, `assert`, and `elements`. We will discuss the `url` and `assert` methods now. The `elements` method will be discussed in more detail below.

The `url` Method

The `url` method should return the path of the URL that represents the page. Dusk will use this URL when navigating to the page in the browser:

```
/**
 * Get the URL for the page.
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

The `assert` Method

The `assert` method may make any assertions necessary to verify that the browser is actually on the given page. Completing this method is not necessary; however, you are free to make these assertions if you wish. These assertions will be run automatically when navigating to the page:

```
/**
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

Navigating To Pages

Once a page has been configured, you may navigate to it using the `visit` method:

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

Sometimes you may already be on a given page and need to "load" the page's selectors and methods into the current test context. This is common when pressing a button and being redirected to a given page without explicitly navigating to it. In this situation, you may use the `on` method to load the page:

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
        ->clickLink('Create Playlist')
        ->on(new CreatePlaylist)
        ->assertSee('@create');
```

Shorthand Selectors

The `elements` method of pages allows you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Now, you may use this shorthand selector anywhere you would use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

Page Methods

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  string  $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
                ->check('share')
                ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed to the page method:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

Components

Components are similar to Dusk's "page objects", but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generating Components

To generate a component, use the `dusk:component` Artisan command. New components are placed in the `test/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to

manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```php
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * Select the given date.
     *
     * @param  \Laravel\Dusk\Browser  $browser
     * @param  int  $month
     * @param  int  $day
     * @return void
     */
    public function selectDate($browser, $month, $day)
    {
        $browser->click('@date-field')
                ->within('@month-list', function ($browser) use ($month) {
                    $browser->click($month);
                })
                ->within('@day-list', function ($browser) use ($day) {
                    $browser->click($day);
                });
    }
}
```

Using Components

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```php
<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
```

```
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->within(new DatePicker, function ($browser) {
                        $browser->selectDate(1, 2018);
                    })
                    ->assertSee('January');
        });
    }
}
```

Continuous Integration

CircleCI

If you are using CircleCI to run your Dusk tests, you may use this configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
version: 2
jobs:
    build:
        steps:
            - run: sudo apt-get install -y libsqlite3-dev
            - run: cp .env.testing .env
            - run: composer install -n --ignore-platform-reqs
            - run: npm install
            - run: npm run production
            - run: vendor/bin/phpunit

            - run:
                name: Start Chrome Driver
                command: ./vendor/laravel/dusk/bin/chromedriver-linux
                background: true

            - run:
                name: Run Laravel Server
                command: php artisan serve
                background: true

            - run:
                name: Run Laravel Dusk Tests
                command: php artisan dusk

            - store_artifacts:
                path: tests/Browser/screenshots
```

Codeship

To run Dusk tests on Codeship, add the following commands to your Codeship project. These commands are just a starting point and you are free to add additional commands as needed:

```
phpenv local 7.2
cp .env.testing .env
mkdir -p ./bootstrap/cache
composer install --no-interaction --prefer-dist
php artisan key:generate
nohup bash -c "php artisan serve 2>&1 &" && sleep 5
php artisan dusk
```

Heroku CI

To run Dusk tests on Heroku CI, add the following Google Chrome buildpack and scripts to your Heroku `app.json` file:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /de
      }
    }
  }
}
```

Travis CI

To run your Dusk tests on Travis CI, use the following `.travis.yml` configuration. Since

Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use `php artisan serve` to launch PHP's built-in web server:

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest
  - php artisan key:generate

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 htt
  - php artisan serve &

script:
  - php artisan dusk
```

GitHub Actions

If you are using Github Actions to run your Dusk tests, you may use this configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
name: CI
on: [push]
jobs:

  dusk-php:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: mysql --user="root" --password="root" -e "CREATE DATABASE my-database
      - name: Install Composer Dependencies
        run: composer install --no-progress --no-suggest --prefer-dist --optimize-
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &
      - name: Run Laravel Server
        run: php artisan serve > /dev/null 2>&1 &
      - name: Run Dusk Tests
        run: php artisan dusk
```

In your `.env.testing` file, adjust the value of `APP_URL`:

```
APP_URL=http://127.0.0.1:8000
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.