How to Use Laravel Dusk for Browser Testing

End-to-end browser testing ensures that your application works exactly as a user would experience it. Laravel Dusk provides a simple API to automate browser actions like clicking links, filling forms, and asserting page content — all without writing raw Selenium code. In this article, we'll set up Dusk, write real tests for authentication and forms, handle JavaScript interactions, capture screenshots and logs, and explore best practices for organizing browser tests in large projects.

Installing and Setting Up Laravel Dusk

```
composer require --dev laravel/dusk
php artisan dusk:installCode language: Bash (bash)
```

After installation, Dusk creates a `tests/Browser` directory with example tests, a `DuskTestCase.php` base class, and updates your `phpunit.xml` to include the Dusk environment. Run Dusk tests with `php artisan dusk`. The package ships with a bundled ChromeDriver binary for headless or full-browser testing.

Writing Your First Dusk Test

```
// tests/Browser/ExampleTest.php
namespace Tests\Browser;

use Laravel\Dusk\Browser;
```

```
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    public function test_homepage_loads(): void
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                ->assertSee('Welcome')
                ->assertTitleContains('MyApp');
        });
    }
}Code language: PHP (php)
```

This simple test checks that the homepage loads, the word “Welcome” is visible, and the page title contains “MyApp.” Dusk uses a fluent API to chain actions and assertions.

Testing Authentication Flows

```
// tests/Browser/LoginTest.php
namespace Tests\Browser;

use App\Models\User;
use Laravel\Dusk\Browser;
use Tests\DuskTestCase;

class LoginTest extends DuskTestCase
{
    public function test_user_can_login(): void
    {
        $user = User::factory()->create([
            'email' => 'dusk@example.com',
            'password' => bcrypt('secret'),
```

```
        ]);

        $this->browse(function (Browser $browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'secret')
                ->press('Login')
                ->assertPathIs('/dashboard')
                ->assertSee($user->name);
        });
    }
}Code language: PHP (php)
```

This Dusk test creates a user with a factory, fills in the login form, presses the login button, and verifies redirection to the dashboard with the user's name visible. This mimics the exact user workflow in the browser.

Handling JavaScript and Vue/React Components

```
$this->browse(function (Browser $browser) {
    $browser->visit('/posts/create')
        ->type('title', 'Dusk Post')
        ->type('content', 'Testing with Dusk.')
        ->check('published')
        ->click('@save-button') // using dusk selectors
        ->waitForText('Post created')
        ->assertSee('Dusk Post');
});Code language: PHP (php)
```

Dusk works seamlessly with JavaScript-heavy UIs built in Vue or React. Use `waitForText`, `waitFor`, and `@dusk` selectors in Blade to target specific elements: `<button dusk="save-button">Save</button>`. This ensures reliable tests even with dynamic content.

Capturing Screenshots and Console Logs

Debugging browser tests is easier when you can see what happened at failure time. Laravel Dusk automatically captures screenshots on failure and stores them in `tests/Browser/screenshots`. You can also capture them manually or store console logs for deeper insights.

Manual Screenshots

```
$this->browse(function (Browser $browser) {
    $browser->visit('/profile')
        ->screenshot('profile-page'); // saves to
tests/Browser/screenshots
});Code language: PHP (php)
```

Manual screenshots are useful in complex flows when you want to verify specific steps. Screenshots are stored as PNG files and can be viewed anytime after test runs.

Console Logs

```
$this->browse(function (Browser $browser) {
    $browser->visit('/dashboard');

    $logs = $browser->driver->manage()->logs()->get('browser');
    foreach ($logs as $log) {
        dump($log->getMessage());
    }
});Code language: PHP (php)
```

Accessing the underlying WebDriver allows you to retrieve JavaScript console logs. This is invaluable when diagnosing Vue/React errors or failed AJAX calls that might not show in the DOM.

Video Recording (Third-Party)

While not built into Dusk, you can run Dusk inside a container with tools like Selenium Grid + ffmpeg to record videos of test runs. This is useful in CI pipelines for diagnosing flaky tests.

Organizing Browser Tests

For large projects, structure your browser tests just like your app's domains:

```
tests/Browser/
├─ Auth/
│   ├─ LoginTest.php
│   ├─ RegisterTest.php
├─ Blog/
│   ├─ CreatePostTest.php
│   ├─ EditPostTest.php
├─ Components/
│   ├─ ModalTest.php
│   ├─ PaginationTest.phpCode language: Bash (bash)
```

This folder structure keeps test coverage modular and easier to navigate. For reusable actions, create custom `Page` objects with Dusk's `php artisan dusk:page` command.

Database and Session Handling

```
// Using DatabaseMigrations trait in a Dusk test
use Illuminate\Foundation\Testing\DatabaseMigrations;

class RegisterTest extends DuskTestCase
{
    use DatabaseMigrations;

    public function test_user_can_register(): void
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/register')
                ->type('name', 'Test User')
                ->type('email', 'newuser@example.com')
                ->type('password', 'password')
                ->type('password_confirmation', 'password')
                ->press('Register')
                ->assertPathIs('/dashboard');
        });
    }
}Code language: PHP (php)
```

Use `DatabaseMigrations` or `RefreshDatabase` traits in your Dusk tests to start with a clean database every time. This ensures tests don't bleed state across runs.

Running Tests in CI/CD Pipelines

```
# GitHub Actions example for running Laravel Dusk
jobs:
  dusk-tests:
    runs-on: ubuntu-latest
```

```
services:
  mysql:
    image: mysql:8
    env:
      MYSQL_DATABASE: laravel
      MYSQL_ROOT_PASSWORD: root
    ports:
      - 3306:3306
steps:
  - uses: actions/checkout@v4
  - uses: shivammathur/setup-php@v2
    with:
      php-version: '8.3'
  - run: composer install
  - run: php artisan migrate
  - run: php artisan serve & # run server in background
  - run: php artisan duskCode language: YAML (yaml)
```

Dusk integrates with CI tools like GitHub Actions, GitLab CI, or Jenkins. Use headless mode for speed, or full browser mode when debugging with screenshots (`--env=chrome`).

Feature Tests vs Browser Tests

Feature tests and Dusk browser tests often overlap, but they serve different purposes. Here's a comparison to decide when to use each:

Aspect	Feature Tests	Dusk Browser Tests
Scope	HTTP layer, controllers, database	Full browser (DOM, JS, CSS, user flows)
Speed	Fast (no browser overhead)	Slower (ChromeDriver involved)
Best for	API responses, redirects, auth rules	Forms, clicks, Vue/React interactions
Setup	No extra drivers	Requires ChromeDriver / headless setup
Example	Assert JSON returned from API	Click "Login" and check UI redirect

Use Feature Tests for logic and data integrity, and Browser Tests for validating the actual user experience in a browser.

Wrapping Up

Laravel Dusk makes browser testing straightforward by combining expressive syntax with ChromeDriver. We covered installation, writing tests, handling authentication, JavaScript interactions, capturing screenshots, console logs, organizing test suites, database resets, and CI/CD integration. With Dusk in place, you can confidently verify your app's real user experience and debug failures effectively.

What's Next

Expand your testing skills with these guides:

- Testing Laravel Applications with PHPUnit
- How to Write Feature Tests in Laravel for APIs
- Using Laravel Factories and Seeders for Test Data