A Little Bit of Laravel

Three weeks of daily tips to make small but noticeable improvements to your apps

AARON SARAY & JOEL CLERMONT

Table of Contents

Welcome

Hey there! Thanks for taking the time to download *A Little Bit of Laravel*, our collection of tips to help make small but noticeable improvements to your app.

This book was written by Joel Clermont and Aaron Saray. Combined, we have over 4 decades of experience programming in web and open source technologies. We've built many Laravel projects over the years and discovered a number of tips and tricks that have made our projects' performance, security and coding experience better.

We're excited to share these with you!

How to Use This Book Effectively

You're already busy working on your app, so we planned these chapters as simple 1 or 2 page chunks for you to digest quickly.

To get the most benefit from this book, we encourage you to resist the urge to speed-read or just scan through it. Instead, read only **one chapter a day** and then see if there's a way to apply what you've learned to your app. We're not asking for a full rewrite, just a few small tweaks to make everything better.

One tip a day for the next 3 work weeks - you can do this! Let's go.

Name Your Routes for Your Future Self

Has this happened to you?

- A new developer joins the team and adds new routes. In our project, we use plural nouns, but they created a bunch of singular ones
- The marketing department requires that you change all of the URLs because they're doing branding changes
- You've changed the nesting of resources, and now you've got to go find every single hard-coded URL and update them with their parent directory

We can guarantee you if they haven't already happened, they will happen to you in the future. How do we make this easier on ourselves?

Named routes!

When you name a route, and then use the `route()` helper, you now have a single place of authority and responsibility for the URLs: your route definitions file.

You will go from something like this:

```
return redirect('/welcome');
```

To this:

```
return redirect(route('first-time-visitor-splash'));
```

Not only will this have all URL construction done based on one source of truth, you also will receive an error from Laravel when you refer to a route that doesn't exist. No more dead links! (Bonus: some IDEs will even auto-complete route names for you.)

To do this, it's simple, just add a call to name() on your route definitions. For example:

```
Route::view('/welcome', 'welcome')->name('first-time-visitor-splash');
```

Laravel also automatically defines names on resourceful route definitions. It provides route name prefix options on route groups as well.

Things to Consider When Validating String Length

One important reason to write tests is to gain confidence that our code works as expected. Many developers focus on the “happy path”, writing tests to make sure all the right things happen as people use our application.

But what about testing when things “go wrong”? For example, what if a user does something unexpected? It would also be great to have confidence we handle those scenarios gracefully and provide useful feedback to the user.

Testing our validation logic can give us this confidence! Let’s consider a text field on a form and a few types of tests we could write.

Here’s what our form validation rules look like:

```
$rules = [
  'title' => [
    'required',
    'max:255',
  ],
  'description' => [
    'required',
    'max:65535',
  ],
];
```

Here is how we can test the required field validation:

```
$values = [];

$response = $this->post(route('route-name'), $values);

$response->assertSessionHasErrors([
   'title' => 'The title field is required.',
   'description' => 'The description field is required.',
]);
```

Here is how we can test the maximum length validation

```
$values = [
   'title' => str_repeat('a', 256),
   'description' => str_repeat('a', 65536),
];

$response = $this->post(route('student.offerings.store'), $values);

$response->assertSessionHasErrors([
   'title' => 'The title may not be greater than 255 characters.',
   'description' => 'The description may not be greater than 65535
characters.',
]);
```

Note the use of `str_repeat`. This makes it very easy to scan and read as compared to putting a 256 character string literal in your test.

As we've shown these tests can be grouped together. We often have one test that checks all required fields and one test for all length-validated fields, and so on. This gives you good confidence without growing your test file any larger than necessary.

Remove Complex Decisions from Your Blade Files

Blade files allow for logical decisions using directives like `@if` or `@isset`. But, we need to make sure to not abuse these by putting too much logic into our Blade files.

The Blade files are the "view" of your MVC-structured application. This means that they should primarily deal with displaying information. Business rules and domain knowledge should have been already decided in a previous layer.

While some data you might send to a view seems obvious, like boolean or string values from a model, others aren't as cut and dried. In that case, it's best to calculate the decision outside and pass in the result to the view.

Let's see an example.

Imagine you have a view that needs to know if the user is a premium customer and if they've set up their billing account. We have a flag on the user to indicate if they're a premium account. We also use Stripe for billing, so a user will have a Stripe customer token on their record if they're all set up.

You might be tempted to do something like this in your controller:

```
return view('the.view.here', [
  'user' => $user,
]);
```

If we're going to get information from the user, why don't we just send in the whole user? Well that's bad for a number of reasons. First, we don't want to send extra information into a view - just in case we were to accidentally display it (weirder things have happened!). And second, it requires our view to understand the construction of the `$user` object. That's too much responsibility and too highly coupled. What if we change the `User` model? We have to now look in all controllers and possibly all views? No thank you!

So what about this?

```
return view('the.view.here', [
  'is_premium' => $user->is_premium,
  'stripe_token' => $user->stripe_token,
]);
```

In your Blade file, you might do something like this:

```
@if($stripe_token)
  <h2>Billing Section</h2>...
```

Now, we're still playing with fire! We're passing a secret to the view, and just hoping we never display it. Also, if we change the billing provider, we have to change the name of this variable. Or worse, leave it the same and have its value mean something different.

Let's do this instead:

```
return view('the.view.here', [
  'is_premium' => $user->is_premium,
  'has_billing_account' => !empty($user->stripe_token),
]);
```

Now, we can check the Blade file like this:

```
@if($has_billing_account)
  <h2>Billing Section</h2>...
```

We haven't passed any secrets to our Blade file, it's easier to read, and doesn't require our Blade template to understand the details of the domain objects. Success!

The Benefits of Resourceful Routes

One theme in these tips you've likely picked up on at this point is our love of consistency. With so many different opinions and ways of organizing your code, it's nice to have one consistent way used within your application.

Resourceful controllers are an example of this, and this tip will focus on the benefits of using the `Route::resource` helper in your routes files.

The benefit when using this helper is that you are adopting Laravel's conventions for resourceful controllers. No more controllers with 20 custom methods! This also means you have consistency in at least two levels of your app: routes and controllers. If you're using policies, this convention now works across three levels.

Second, this helper reduces the size of your routes file. As your app grows, the routes file can get out of hand quickly. This method will condense as many as 7 separate route definitions into one simple line:

```
Route::resource('widgets', WidgetController::class);
```

Finally, the helper can be easily customized when needed. Sometimes, a resource doesn't need all conventional methods. For example, maybe you never allow a resource to be deleted:

```
Route::resource('widgets',
WidgetController::class)->except(['destroy']);
```

Or let's say your resource is read only:

```
Route::resource('widgets', WidgetController::class)->only(['index',
'show']);
```

Using Console Routes for Ad-hoc Testing

Most developers reach for Tinker when they want to run something interactive, in real time, without loading a complex webpage. This is convenient since you have the fully configured Laravel app at your fingertips.

But, what about times when you want to experiment and juggle the order of your method calls, your dependencies, or anything else? It can get tedious rebuilding these things in an interactive shell. Or even worse, you can lose track of what you did that finally worked.

Enter closure-based console commands. The `routes/console.php` file contains an example command to get you started. You can add a new one by temporarily defining a new route and closure. Then, you can easily run your command over and over with the `artisan` tool while making incremental changes.

For example, let's say we wanted to test a complex third party service integration with some data from our application. We might set something like this up:

```
Artisan::command('test', function () {
  $service = app()->make(\App\Services\ThirdPartyService::class);
  $myModel = \App\Models\MyModel::findOrFail(42);
  $data = [
    'id' => $myModel->id,
    'label' => $myModel->label,
  ];
  $service->sendToEndPoint($data);
});
```

Then, you can run your command:

```
artisan test
```

Next time you reach for Tinker to whip up a quick proof of concept function, give this technique a try instead.

Global Scopes as Reusable Classes

Eloquent provides two methods to modify queries for a model: local scopes and global scopes. In this tip, we're going to focus on global scopes which are query modifications that are applied to an eloquent model every time it is retrieved.

When writing a global scope, you can either add it directly to the class' `booted()` method as an anonymous function or you can register it with a class. The Laravel documentation describes writing a class-based global scope that is pretty specialized for a unique case in an application. But, don't let that limited example fool you - this feature is very powerful.

One of the things that we've used global scopes for quite often is applying an order to the result set of a model. Depending on the model, you might have a specific ordering that makes sense. Invoices might be ordered by created at or sent at dates. Cities might be ordered by their name field.

If you find you're writing a lot of similar global scopes like this, it's time to invest in a good reusable class for applying your global scopes. In this example, let's add a single line to the model's `booted()` method to order its results by a field and direction unique for this model.

Here's how it looks in two different models:

File: app\Models\City.php

```
<?php
namespace App\Models;
use App\Scopes\OrderBy;

class City extends Model
{
  protected static function booted()
  {
    static::addGlobalScope(new OrderBy('name', 'asc'));
  }
  ...
```

File: app\Models\Comment.php

```
<?php
namespace App\Models;
use App\Scopes\OrderBy;

class Comment extends Model
{
  protected static function booted()
  {
    static::addGlobalScope(new OrderBy('created_at', 'desc'));
  }
  ...
```

Now, what does this reusable global scope look like? Let's check it out.

File: app\Scopes\Orderby.php

```
<?php
namespace App\Scopes;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class OrderBy implements Scope
{
  protected $column;
  protected $direction;

  public function __construct(string $column, string $direction = 'asc')
  {
    $this->column = $column;
    $this->direction = $direction;
  }

  public function apply(Builder $builder, Model $model)
  {
    $builder->orderBy($this->column, $this->direction);
  }
}
```

Now, we have a global scope for ordering our results that truly is reusable. This is just one example. Any time you find yourself writing multiple variations of a very similar scope, consider this technique.

Quickly and Easily Reject the Most Common Passwords

A common way applications get “hacked” is that users reuse the same password among services. So, if they fall victim to a phishing attack for one site, the bad guys now have their password for many sites.

While we might joke that that sounds like a user problem, not ours, that’s not true. The story never is “users reused passwords leading to breach in software.” Instead, it’s sensationalized to say “LaravelTipsReader’s Website Hacked in Crime of Century!” or something like that. Your site gets blamed even if it’s not directly your fault.

Whether you think it should be your responsibility or not, stopping password reuse on your website should be a priority now. One way sites do this is by requiring more and more complex password syntaxes. The effectiveness of this and its corresponding user experience is a whole other conversation we’re not going to have in this quick tip.

Instead, let’s focus on a quick win. Each year, security companies publish a list of the top 25 most common, most reused passwords. This sounds silly that anyone would use them, but they do! In our Laravel apps, we can easily make sure that no one uses these passwords. That seems like the least we can do to protect our users and the site’s reputation.

Let’s get started.

Step 1 - search “top 25 passwords of ####” where #### is the current year. Or, you might make a combined list of a couple years. Point is, you want to develop a list of common passwords that you want to immediately reject.

Step 2 - make a Laravel Validation Rule with the responsibility to reject any of those common passwords. Don’t get fancy like hitting an API to retrieve a list or importing a CSV in real time. Just hard-code the values in like this:

File: app/Rules/RejectCommonPasswords.php

```
<?php
namespace App\Rules;
use Illuminate\Contracts\Validation\Rule;

class RejectCommonPasswords implements Rule
{
  public function passes($attribute, $value)
  {
    return !in_array($value, [
      '12345',
      '123456',
      '123456789',
      'test1',
      //... a bunch more here
      'Password',
      'maria',
      'lovely',
    ]);
  }

  public function message()
  {
     return trans('validation.reject_common_passwords');
  }
}
```

Step 3 - add a translation to all of your language files under the `validation.reject_common_passwords` key. Or, you can simply hard-code a message in this method. Use something clear, but not overly detailed: “Please choose a stronger password.” Be concise, but not insulting.

Step 4 - add this rule to every password field in your app. These probably are your sign up and reset password form requests/validation configurations. For example, your password validation might look like this:

```
'password' => [
  'required',
  'string',
  'min:8',
  new RejectCommonPasswords(),
]
```

Or, you can combine this with the password validation rule in Laravel 8:

```
'password' => [
  Password::min(8)->letters()->numbers(),
  new RejectCommonPasswords(),
]
```

Note that this is not a replacement for the `Password::uncompromised()` method. But, if you're in a situation where you need to eliminate your calls to a third party, the reject common passwords rule can help.

Now, you’re doing your part to help make your application more secure for your users.

Keeping Authorization Logic Consistent

Consistency is extremely valuable in a code base, whether you're a solo developer or working on a large team. By establishing some guidelines as to where certain types of logic belong in your codebase, you make it much easier to maintain your application.

For example, where should you organize your authorization logic? How should you enforce these authorization checks? You could do it in a form request, inside a controller action, within a middleware, or even inside of a view.

Let's start with where to put the authorization logic. We like to use Laravel policies whenever possible. The policy provides a central place to store all authorization logic for a specific resource. Conventions exist for all the typical actions you want to perform and these map cleanly onto the conventions around resourceful controllers. And if you're not using resourceful controllers yet, you are still able to add custom methods to your policy as needed.

As a bonus, if you integrate Laravel Nova into your application, it will leverage those same policies you already built for your main application.

If you're sticking with the resourceful conventions, you can authorize a whole controller with a single line in the constructor:

```
public function __construct()
{
  $this->authorizeResource(Post::class, 'post');
}
```

If you're not using resourceful methods, you can check inside the controller action with one line:

```
$this->authorize('customMethod', Post::class);
```

One additional benefit of using these helpers is that the correct HTTP status code is generated for you if the authorization fails.

Enforcing authorization logic is best done as early in the request lifecycle as you can. Sometimes you can protect a whole group of routes with a middleware check, like when you're using a role and permissions system:

```
Route::middleware('can:manage-posts')->group(function () {
   Route::resource('posts', PostsController::class);
  Route::get('posts/{post}/approve-post', ApprovePostController::class);
});
```

Using a Bootstrap File for PHPUnit Can Make Life Easier

The standard install of a Laravel application contains a PHPUnit test suite that is set up pretty nicely. The application loads up the Composer autoload functionality, uses the Abstract Test Case to register traits, and builds up an encapsulated Laravel environment.

What happens if you want to add some other set up into that process, though? You have the standard PHPUnit `setUp()` and `setUpBeforeClass()` methods, but those are run on each test and each test class, respectively. What if you just need to bootstrap a few more things before your entire set of tests kick off?

The unit test process gets configured using the `phpunit.xml` file in the root of your project. In the root `phpunit` element of the XML, there is an attribute called `bootstrap`. By default, this is loading your Composer autoloader to complete bootstrapping the test environment.

In our Laravel projects, we use the Mockery library for mocking items. While we could use static methods on fully qualified classes, we prefer to use the global method helpers in our test suite. In order to use the method `mock()` for example, I need to run the `Mockery::globalHelpers()` method.

First, I'm going to modify `phpunit.xml`, find the `phpunit` root element and modify the `bootstrap` attribute like this:

```
<phpunit bootstrap="tests/bootstrap.php" ...>
```

Now, I'll create the file with the following content:

File: tests/bootstrap.php

```
<?php
require __DIR__ . '/../vendor/autoload.php';

Mockery::globalHelpers();
```

Now, when PHPUnit tests begin, the bootstrap process calls our new bootstrap file. The first thing it does is load the Composer autoloader. Then, it calls the method to register the Mockery global functions. If we wanted to add more bootstrapping code for our test environment, we could continue to expand this file.

Eloquent Methods to Encapsulate Creation Logic

There are a number of ways to create new Eloquent models. The one we reach for the most is calling the static `create()` method. For example, to create a new `Car` model, the code in the controller might look like this:

```
$myCar = App\Models\Car::create([
  'make_id' => $ford->id,
  'model_id' => $explorer->id,
  'year' => '2019',
]);
```

Great! But, what about if our creation of a model gets more complicated? For example, let's say we have this in our controller:

```
$owner = $tradeIn->owner;
$address = $owner->homeAddress;
$insuranceRegion = Region::where('city', $address->city)
  ->where('state', $address->state)
  ->firstOrFail();

$myCar = App\Models\Car::create([
  'make_id' => $ford->id,
  'model_id' => $explorer->id,
  'year' => '2019',
  'owner_id' => $owner->id,
  'insurance_region_id' => $insuranceRegion->id,
]);
```

In this contrived example, our car model requires not only identifiable information about the vehicle, but information about the owner as well. It also requires we determine what the insurance region is for that automobile.

Now, this isn't so bad at first glance, but can we make it encapsulated and easier to use? Normally, we wouldn't recommend creating a method just for a single time's sake, but this is different. Here, we want to encapsulate some required business needs into one method so that we guarantee they happen. It makes it easier to read the controller and less likely we'll miss something if we use it again. Plus, it provides a way to make a more self-documenting bit of code.

Let's change our code to this:

```
$myCar = App\Models\Car::createFromTradeIn(
  $tradeIn,
  $ford,
  $explorer,
  '2019'
);
```

Then, in our Car model, we can have a method like this:

```
public static method createFromTradeIn(
  Car $tradeIn,
  CarMake $make,
  CarModel $model,
  string $year
) {
  $owner = $tradeIn->owner;
  $address = $owner->homeAddress;
  $insuranceRegion = Region::where('city', $address->city)
    ->where('state', $address->state)
    ->firstOrFail();

  return static::create([
    'make_id' => $make->id,
    'model_id' => $model->id,
    'year' => $year,
    'owner_id' => $owner->id,
    'insurance_region_id' => $insuranceRegion->id,
  ]);
}
```

Now, we can call the more obvious method `createFromTradeIn()` and the logic required to transfer a car's information to a new car is all encapsulated.

This is one of those tips that you need to be careful with, however. Don't start creating too many single-use methods just because it seems like a good idea. Only create methods like this where the set up takes a few steps to retrieve and manipulate the data.

Naming Events and Listeners

It's often said that there are two hard problems in programming: cache invalidation, naming things and off-by-one errors.

With that in mind, let's talk about naming things, specifically events and listeners. Naming events and listeners can be hard. However, there's a simple trick to doing this easily.

Name the event after what happened. Name the listener after what it's tasked to do.

While listeners can listen to a specific event, you're not required to limit them to one-to-one relationships. You can have multiple events that all have a single listener. Or you can assign a listener to multiple events. As a bonus, with this naming convention, you're more likely to write simpler code that only does the thing the class is named.

Let's look at an example:

`App\Events\UserConfirmedEmail` is issued when a user clicks a confirmation link after they've signed up themselves.

`App\Events\AdminCreatedAccount` is issued when an admin creates a fully activated user account (no email confirmation is required because the admin knows the information is legitimate).

`App\Listeners\DispatchWelcomePackage` is a listener that sends a welcome email with attachments and other information for your product.

Now, look at our configuration for our `EventServiceProvider`. We can reuse this listener and it makes complete sense.

```
protected $listen = [
  App\Events\UserConfirmedEmail::class => [
    App\Listeners\DispatchWelcomePackage::class
  ],
  App\Events\AdminCreatedAccount::class => [
    App\Listeners\DispatchWelcomePackage::class
  ],
];
```

This would be more confusing if we had named our listener something like `HandleUserConfirmedEmail` — it'd be confusing as to why that is registered under an event that was not email confirmation.

Because of listener reuse, however, type-hinting an event and using automatic listener registration will not work and should be avoided.

Using Composer Scripts for Common Tasks

Most PHP projects use the Composer package manager to manage dependencies and Laravel is no exception. You might even have installed Laravel the first time using Composer. Besides package management, Composer adds a number of other features to your project. One of them is the ability to run scripts in your current project.

Projects like Laravel make use of Composer's lifecycle hooks by registering scripts under specific hook names like `post-create-project-cmd` or `post-autoload-dump`. But, the scripts section is not limited to just lifecycle hooks. You can also define other commands that you might want to use during development or continuous integration.

When you define a script under the `scripts` key of the `composer.json` file, it runs in the current working directory with access to everything in your current path. In addition, it also adds your project's `vendor/bin` directory at the beginning of your path resolution. This means you can use your project's local dependencies in your Composer scripts.

To see all of the commands that Composer can execute, you can simply run `composer` with no arguments and options. You'll see a mix of built-in commands and non-lifecycle scripts. You can describe these scripts in the `scripts-descriptions` key.

Let's take a look at an example.

File: composer.json

```
{
  "scripts": {
    "tests": [
      "phpunit --colors=always"
    ]
  },
  "scripts-descriptions": {
    "tests": "Runs PHPUnit tests with console colors"
  }
}
```

To run PHPUnit tests in your project, it's now as easy as `composer tests`. When you see the list of Composer commands, the tests command will now be properly described. Running this Composer script is the equivalent of running this command in your project: `vendor/bin/phpunit --colors=always`

Oh, and one last thing. The `scripts` elements are all arrays. This means you can register multiple commands to run in order that can all be invoked as a single Composer script.

Testing a Scheduled Command On Demand

There are differences between running console commands directly compared to when they're run with the Laravel scheduler. But, because of their schedules, these commands can be hard to test locally in this configuration.

If your command runs every minute, you could execute it by running `artisan schedule:run` locally. However, unless you're exceptionally good at running this command at a specific time - or you're super patient and can wait hours - you might find other commands harder to test.

To handle this, the Laravel Scheduler exposes a useful method to test any scheduled command regardless of the current time.

```
artisan schedule:test
```

When you run this command, you'll get a list of scheduled commands. Pick the one you'd like to run and it will be run on demand. This command will bypass the truth-test constraints on your command definition, however. Or to say another way, even if your schedule has `when()` and `skip()` declarations that would resolve to restrict the command from running, this method will run them anyway.

Covering Value Ranges in Your Tests

One big benefit of tests is that they increase confidence in our code. We write tests while we're building a feature to confirm it works as expected. We run these tests every time we make a code change to check if we broke anything that was already working. Knowing what to test is an important skill and today's tip will point out one area you may not have considered.

Let's picture a relatively simple form with some text inputs that will get persisted to a database. You'll create a migration that defines the field definition in the table, a form request which validates the maximum length of that field, and usually some HTML input properties to also enforce the length in the browser.

Generally, we would write a test for the "happy path", where the form submission will succeed and all the inputs will pass validation. We'd also write some failing tests to confirm that our form request validation is catching missing required fields or fields that are too long to be stored in the database.

But consider one additional test to write: Making sure you can submit a value that is exactly the longest allowed length. So if your database field allows 1000 characters, make sure your happy path is storing a 1000-character value successfully.

This actually bit us in production when we intended to allow 1000 characters, set up the validation and HTML form attributes to match, but then forgot to set the length in the database migration. Instead it used the default length, so the first time someone typed in more than that default length in production, the request failed with a `Data too long for column` exception from MySQL.

Lesson learned. Now anytime there's a non-default length for a database field, we make sure to cover the max length in my happy path test. It doesn't take any extra time, and it gives us just a bit more confidence our code is working as expected.

Make Upgrades Easier by Simplifying Configuration Files

Upgrading to the newest Laravel version can be an exciting time. Hooray for all the new features! Usually the upgrade process is fairly straightforward, but today's tip will help you avoid some potential problems.

First, it's important to clarify the types of things that get upgraded with a Laravel project upgrade. Most of the code changes come via package updates to the framework itself. These are relatively automatic. Composer handles that for you, and you just need to make sure your tests still pass and things work as expected.

The other type of change is a little bit more complicated: changes inside your app folder. For example, looking at the Laravel 8.0 release, here are just a few of the files changed inside your application code:

- `app/Exceptions/Handler.php`
- `app/Http/Kernel.php`
- `app/Providers/RouteServiceProvider.php`
- `config/auth.php`
- `config/queue.php`

This tip will focus on just the configuration files. By keeping Laravel's configuration files as unchanged as possible, you can make these version upgrades much easier to manage. How can we do this? There are two main strategies.

1. Leverage environment variables

Many of the configuration settings you'll need to change first check for a specific environment variable before falling back to a default specified in the config file. Wherever possible, use these environment variables to customize the config. Plus, as a bonus, you're on your way to an OWASP secure project!

2. Create your own custom config file

The config files you get with the framework are only a starting point. You can create your own file for things not covered by existing configuration. I typically like to name this file after my application, or just call it something generic like `config/custom.php`. Inside this config file, you can do whatever you want and leave it untouched during the next Laravel upgrade.

With these two strategies, you can keep your framework config files as close as possible to their default states. Yes, there will still be some things you may need to modify, but by keeping the number of changes as low as possible, your `git diff` will be a lot cleaner when merging in upstream Laravel changes, saving yourself some time.

When to Use Single Action Controllers

In a previous tip, we shared our recommendation for using resourceful controllers wherever possible. While it keeps code consistent and easier to navigate, there are some use cases where a resourceful approach just doesn't fit. Instead of bolting on an arbitrary action to a resourceful controller, we can use a single action controller instead.

Just as the name suggests, a single action controller is literally just a controller that has one public function: `__invoke`.

Here are two common scenarios where single action controllers are a good fit:

Content pages

If you have a very simple, static page, you can route directly to a view and skip the controller altogether. But what if your mostly static page also needs some data bound to it? Or what if you have a dynamic page, like a dashboard, that has data bound to it for the logged-in user? A single action controller is a clean way of implementing it.

Workflow logic

Take for example a job application process. The resource is the job application, but when an application is approved or rejected, that isn't the same thing as editing the resource. For one thing, different fields on the job application are likely editable by different roles. The applicant can change most of the information, but they can't change the status to approved. In addition, the workflow around approving an application likely does more than just change a status field to a new value. You might send an email, reject other applications, and so on. Trying to do all these different things in a single resourceful edit/update action will get messy very quickly. A single action controller is a much cleaner fit for the workflow logic.

Using this approach, you may worry that you'll end up with too many controllers, which would pose a different set of problems. While it's true you'll have more controllers, we argue that the end result will be far more manageable than a smaller number of giant controllers with lots of custom actions. If navigation truly becomes a challenge, don't forget you can group single action controllers by feature into namespaced folders to make it easier to navigate.

Nail Down a Flaky Test

A failing test is actually a good thing, since it helped us catch a potential bug before it reached production. But what about a flaky test, that is a test that only fails every once in a while for no apparent reason? Flaky tests are extremely frustrating. They slow us down and erode our confidence in the test suite. They can also be difficult to diagnose.

Today's tip will share a real world example of a flaky test and highlight some tips on how to debug them and confirm they're really fixed once and for all.

This particular flaky test involved sending an SMS notification when someone was awarded a job application. Our test would assert that the right person was notified, that their preferred mobile number was used, and that the expected message was sent. Most of the time, this test would pass, but every 1 out of 100 times, it would fail. Quite annoying!

The first issue to solve is why the test was failing. In our test, we were using Mockery to mock the actual SMS client so we avoided sending real messages during test runs. But because of how Mockery was failing, our test failure gave us no indication as to why it was failing. To get to the root cause, Mockery was temporarily removed from that test, and instead we added some extra code to the class being tested to avoid the external call.

Next, it was time to get the test to fail so I could get at the real error, but how? PHPUnit has a handy command line option `--repeat` which does just what you might expect. So with the new test code in place, this one flaky test was run with `--repeat 1000`.

Hooray, it failed after 100 or so runs! With this temporary test debug code in place, we finally had the root cause: SMS messages get truncated at 160 characters. So if the factory generated a test job with a title just a little bit too long, it would push the message over that boundary, and cause a test failure. Armed with this knowledge, the test was easy to fix.

With the fix made, the temporary debugging code was removed, Mockery was restored and we ran the test with `--repeat 1000` again. This time it got all the way through all 1000 runs without a failure. The flaky test was squashed!

This may seem like a lot of effort for something that doesn't fail all that often, but the investment is worth it. True, a flaky test may fail only 1 out of 100 times, and when it does fail, you can just run the tests one more time to get them to pass, but we recommend you don't live with the pain of a flaky test. Get to the bottom of it and regain confidence in your test suite.

You're Done. What Next?

Thanks for reading our tips book. We hope you've learned a few things along the way and made some real positive changes to your app.

In fact, we'd love to hear about your success stories or things you've learned. You can email us at hello@nocompromises.io or tweet Joel (@jclermont) or Aaron (@aaronsaray) directly.

Keep an eye out for news and updates from us, too. We're launching some cool Laravel learning tools soon and we'd love you to be part of the community.

Want to Learn More?

If you enjoy our style, you might be interested in our other book Mastering Laravel Validation Rules. It's a deep dive into validation rules with real-world examples, hands-on exercises, and a dash of fun.

You'll get **15% off** both the ebook and validation exercises when you use this link (this special link contains the discount code).

---

ps... *when you read a book, do you "hear" a voice for the author? Like, make one up and sort of hear that in your head as you're reading? Or is that just something weird Aaron does... Anyway, if you want to hear the sweet, dulcet tones of the two authors, check out the No Compromises Podcast.*

Written by Aaron Saray and Joel Clermont

This book was written referencing Laravel 8.41.