Code Review for Laragigs

This review was done from the git commit `cac39143d60543d852c4e52b3f17ebd8cf0ea17f` - this may not match the exact current commit on the source project or the forked version.

Quick note about this code review There are a number of good things in this project that I like an agree with. Others there's no reason to comment - maybe it's just a difference of opinion. The goal of this code review is only to point out things that need improvement or are, in my opinion, a better way to do things. I will try to mention things that are common best practices in PHP/Laravel projects and differentiate things that are specific to my own opinion.

This particular project does not have much/any CSS or Javascript. Therefore, this review will primarily be based on PHP and Laravel code. I'll likely not give opinion on HTML structure in most places, either.

Remember to apply the feedback to the entire project. Even if the problem or best practice was only mentioned on one file doesn't mean it doesn't exist in others. It would be a lot of work to point out each one - and I trust reviewers can pattern match and apply the feedback globally.

I want to thank the developer for releasing this open source for others to use and for me to review. You rock.

On to the review content.

PHP Code

composer.json - There are a lot of important sections in the composer file. While each has a different level of importance, all do matter.

- Replace the name of the project. In some instances, this has caused problems with some automated tools. They may believe the project is something else. But in general, this is just nice so that if your project source leaks out, it's clear that it's yours and no one else's. Along those lines, I'd just remove the keywords section and make description very simple.

- Add the Roave Security Advisories package. This specifically introduces conflicts into composer for libraries that have vulnerabilities. That is to say, it will stop the installation of packages that could lead to the application being hacked.
- Use the `scripts` section of the composer file to script out common things that you might do - like Unit test running or code quality tools. For example, I may write a script entry for each of tests, phpstan and code sniffer. Then, I'll create a `ci` script entry that I run before I merge things - or with github actions - to run all three one after another - to confirm the project is still up to standards. It helps to reduce the amount of keystrokes and reduce the memory of individual steps of process I need to know.
- This is a contrary opinion to a lot of the community - but I tend to move `laravel/tinker` to `require-dev`. I don't want people using Tinker in production systems. If you want to do something, you can query the database with a read-only user, or you can write a one-time script. The scripts will then be version controlled, so we'll always know what happened. I've seen too many times where people blow away production data with Tinker, not realizing what environment they're in.

data.php - not sure what this was for, but it should be removed. Remember, you can put files anywhere you want, and then just Git stash them for later if you really need to. Do not commit things that have no place in the project.

routes/api.php - Remove an end point declaration like this - or even comment it out. You haven't built out your API, so therefore you probably haven't thought through all of the specifics of what would be returned about a user like this. Sure, it's probably fine, but why leak information? "Why would that matter" is the clarion call of a lot of unforeseen security holes. This goes back to removing code that's not in use. Even though you may not have touched the code in your project, you're responsible for all of it - even if it was autogenerated.

routes/console.php - Same thing here, remove the inspiring quote command. You're not using this anyway, right? :) - But seriously, remove code you're not using. I find the console php file is good for temporary commands - but for longer-lived commands, you're going to make full console command files anyway.

routes/web.php - Sometimes this feedback is aimed at a specific file or line. That doesn't mean that's the only place that that happens or is applicable. Please review the entire file for the pattern identified by the feedback line.

First an opinion - I tend to import only `App\Http\Controllers` at the top. Then, when you refer to something like `RatioController` it's actually going to be `Controllers\RatioController` instead - but the benefit is two-fold: less imports and

more verbosity without fully qualified names. That's just an opinion thing, though. These route files can balloon to hundreds of lines, and the imports get pretty unmanageable.

Routes tend to have names. You can see your existing ones by running `artisan route:list` - here you'll see you could swap `->get('/login')` to `->get(route('login'))` or `action="/login"` to `action="{{ route('login') }}"`. This way if you always use route names, you get two benefits: 1) you can change the path of urls but not affect your code and 2) you can see where you've tested those routes and programmed in them - because you can always search for the route name - versus some combination of regular expression for complex paths that may contain parameters.

You can group things when they share similar things - like middleware, names or prefixes of the URL. For example:

```
Route::get('/listings/create', [ListingController::class, 'create'])->middleware('auth');

// Store Listing Data
Route::post('/listings', [ListingController::class, 'store'])->middleware('auth');

// Show Edit Form
Route::get('/listings/{listing}/edit', [ListingController::class, 'edit'])->middleware('auth');
```

This could be compressed to something like this:

```
Route::middleware('auth')
    ->prefix('/listings')
    ->name('listings.')
    ->group(function () {
        Route::get('/create', [ListingController::class, 'create'])
            ->name('create');
        Route::post('/', [ListingController::class, 'store'])
            ->name('store');
        Route::get('/{listing}/edit', [ListingController::class, 'edit'])
            ->name('edit');
    });
```

It technically looks like more lines because of the line wrapping - but it's really only a few. What this does is the following:

- first it applies the `auth` middleware to all things in the group

- then it prefixes the URLs with `/listings` - so you don't have to type that each time for the other declarations
- it applies the name `listings.` to everything in the group. So, it adds the declared name to that. So, when you call `->name('create')` it really makes `listings.create`

Laravel has this concept called resourceful controllers. They are a pattern that is created/defined for CRUD actions, that is list, single view, create, update, delete. It handles the form view for create/update - as well as the processing for store/update. When you have something like this listings controller, it's best practice to use a partial of a resourceful controller. So, you get in the pattern of using a `create()` method and a `store()` method - and the rest of them - then when you define them in the routes file, you can do something like `Route::resource('listings', ListingsController::class);` - it will use the predictable methods and generate predictable route names. No need to invent the wheel on each controller then! :)

Finally, it's best to use a Laravel authentication scaffolding like Sanctum for your authentication. You won't have to re-invent this code like what's required in the `UserController::authenticate()`` method.

config/app.php - Probably never going to happen, but someone may refer to the `APP_NAME` somewhere (like a notification). Change the default value to your company name.

database/seeders/DatabaseSeeder.php - a couple things:

- first, don't keep commented code. Git allows you to go into the history and get things if you need them. However, in this case, I'm guessing it has something to do with only wanting to run specific seed data at specific times. In that case, make another seed class, and put it in there, uncommented. Then, do *not* call that seeder from the main `DatabaseSeeder` class - but you can specify it with your artisan command when you need it: `php artisan db:seed --class=MyUniqueSeeder`
- When you have a relationship like `Listing::user()`, you can then use the helper method `for()` on factories. This way you don't have to know the keys or use the `id` property. You can do something like this: `Listing::factory(6)->for($user)->create()`.

app/Providers/AppServiceProvider.php - There is a call to `Model::unguard()` in the `boot()` method. This is a very bad idea. This globally removes some of the protection that Laravel uses to secure your models and data. Guarded data means that Eloquent will not allow bulk data sent to methods like `create()` or `update()` unless it is specified in the `$fillable` property of the model, or - not in the `$guarded` . That doesn't mean that you can't set data individually or using the `forceFill()` method - but those are things you should reach for as a last resort. Security in depth requires guarded models. I suggest

specifying the fillable properties only. Think about it this way: you maybe want to create a blog post. You will allow people to bulk submit title and body, so you allow these to be fillable. You don't want anyone to set the user id field, so that's not listed as something that is accepted. In your code, though, you probably will build the blog post off of the user using the blog posts relationship.

app/Models/User.php - when defining a relationship, like `listings()`, here are two suggestions:

- first, when possible, I suggest type hinting your return types. Some IDE's will be able to understand that the `HasMany` comes from the call to `$this->hasMany()` but to be more precise and make it more obvious, write this like this:

```
public function listings(): HasMany
{
  return $this->hasMany(Listing::class);
}
```

- another thing you'll notice is that there is no `user_id` referenced either. Laravel conventions dictate that the relationship property is always the name of the source class, singular, with `id` suffixed. So, since you're in the `User` model, it will always look for `user_id` on any `HasMany` relationships. Only if this column doesn't match conventions do you need to specify it.

app/Models/Listing.php - a couple items:

- just like the previous model, the `user_id` is redundant in the `BelongsTo` relationship of `user()` - this convention works for all of the relationships
- the fillable property is commented out. I bet this has to do with the unguard command in the app service provider. This property should be defined with the values it currently has in the comment
- the scope should have type hinted parameters to make the queries easier and should not refer to the request. It's already passed in. Also, the query statement is not going to generate what I believe is the expected result. Consider this old code:

```
public function scopeFilter($query, array $filters) {
    if($filters['tag'] ?? false) {
        $query->where('tags', 'like', '%' . request('tag') . '%');
    }

    if($filters['search'] ?? false) {
        $query->where('title', 'like', '%' . request('search') . '%')
```

```
            ->orWhere('description', 'like', '%' . request('search') . '%')
            ->orWhere('tags', 'like', '%' . request('search') . '%');
    }
}
```

The biggest issue here is that the scope or level of all of the where statements. So, this will actually create a query that where's on the tags, where's on the title, and then or where's on the other two. When, in reality, what is probably expected is the first two wheres but the second should be a nested or considering all of the columns. Perhaps this is easier to understand with code:

What it could be updated to:

```
public function scopeFilter(Builder $query, array $filters = []): Builder
{
    if ($tags = Arr::get($filters, 'tags')) {
        $query->where('tags', 'like', "%{$tags}%");
    }

    if ($search = Arr::get($filters, 'search')) {
        $query->where(function (Builder $q) use ($search) {
            $searchTerm = "%{$search}%";
            $q->where('title', 'like', $searchTerm)
                ->orWhere('description', 'like', $searchTerm)
                ->orWhere('tags', 'like', $searchTerm);
        });
    }

    return $query;
}
```

Let's note the following:

- first, the `Illuminate\Database\Eloquent\Builder` is hinted with the incoming `$query` parameter. That way we get a nice set of autocomplete in an IDE and guarantee that we know what we're working with
- next, I set the default value to an empty array by default. Likely you'd have a key of either `tags` or `search` - but just in case you have none, or don't even pass in anything, this will make sure the code doesn't fail.
- Using the `Arr` support helper, we can work with arrays in a unique way. We retrieve the `tags` key from the array, if it exists, otherwise it returns `null` if it's not set. Then, we already have the variable set with the value if it exists.
- In the first query, I'm using a string that has the percent on both sides, and then a rendered variable. Since these commands are all prepared statements, we don't have

to worry that we're introducing any weird queries or security holes with this setup.
- The same thing is repeated for the search term. Instead, the string is built so it can be passed in.
- Note the nested where statement. Without this, you run into problems in rare situations. For example, if you passed in a filter of some tags and then a description, you'd want it to match the tags and then match the description as well. You're 'or'ing the three fields together as an option that has to match if that key exists. Otherwise, in the old system, it would find something with the tags - or - it would find something with the description. It wouldn't be things that are tagged that way AND match the description - which is really what is wanted.

app/Http/Controllers/ListingController.php - the following feedback:

- First, note how this controller has the methods `index`, `show`, `create`, `store`, `edit`, `update`, `destroy` - it basically follows the paradigm of the resourceful controller. So, the feedback about resourceful controller is basically already ready for this one.
- There's a `manage` method. It's best practice to make resourceful controllers. Then, with this extra method, move it to the `__invoke()` method of a new controller called `ListingManageController` or something like that. Invokable controllers are great for one-off things.
- `index()` method:
  - First, I recommend - and this is mainly an opinion and not necessarily a globally accepted best practice - but I recommend injecting the `Request` object and retrieving data off of that instead. Do not use global methods like `request()`. This way, it's clear what request you're working with.
  - Also, just because something is a GET, doesn't mean that the parameters shouldn't be validated. There is no validation on the filter request variables. This should be done, perhaps with a form request, and then you can retrieve the validated data off of that.
  - `6` is a weird pagination number. But in general, magic numbers and strings are bad. That is to say, something that means something but is string or integer constant value can be easily missed. For something like this, a class constant for pagination number should be generated.

```
private const PAGINATION_PAGE = 6;

public function index(ListingIndexRequest $request) {
  $listings = Listing::latest()
      ->filter($request->validated())
      ->paginate(self::PAGINATION_PAGE);
```

```
    return view('listings.index', [
            'listings' => $listings,
    ])
}
```

- `store()` method:
  - glad to see validation. However, there are a few things wrong with it - or things that could be better.
    - First, make it a form request. Then, you can have your validation in a separate class, know it's applied, and not have to worry about a larger controller
    - second, the requirements of each field should be much stronger. Consider the data type (string, int), the length (what if someone enters 300 chars, but your database only handles 255?), things like that. For example, for `title` I'd likely put something like `title => ['required', 'string', 'max:255'],`
    - the logo item is not validated. Even though you deal with it differently (using the `->file()` method) you should still validate that it's a file, what type it is, the size constraints, etc. Especially since it wasn't validated, you could now upload something like a `php` file and that is now in the public folder and can be ran from the internet. Not good. (Validation of images for image mime types is great. Some even run the image through a GD or Imagick filter and re-generate it so it's smaller, or at least a known image. If you had an image with PHP or JS injected, running it through a conversion would remove that.)
  - you should always get the current user off of the request. `$formFields['user_id'] = $request->user()->id.
  - Instead of that, though, I would suggest creating the listing directly from the user - so you don't have to unprotect or fill the user id specially. Do `$request->user()->listings()->create($formFields);` and this won't require you to specify the `user_id` at all. Eloquent just handles that for you.
  - on redirects, if you have routes named, you could do something like `return redirect(route('my-route-name.here'));` and you now don't have to ever search for `/` - you can find where routes are in use by their name.
- `update()` feedback here:
  - first of all, glad to see some authorization logic at top. If you use a form request, you can put that in the `authorize()` method. In addition, if you create policies, that could be done automatically from the controller declaration.
  - there is code duplication between `store` and `update` - I think that could be removed, or when the form requests are created, that will get rid of a lot of that anyway

- on update, what about the old image? do they all just say around forever? or should it be removed when there is an updated image?

app/Http/Controllers/UserController.php - there isn't really much to say about this controller. It's a lot of re-invented wheels. Instead of commenting on this, it'd be best just to use something like Laravel's Auth scaffolding like Sanctum. It writes all of this in a way that's really robust, tested, and still configurable.

View/Blade

All of this feedback is based on the folder structure **resources/views** and won't have that prefix below.

components/flash-message.blade.php - I like the idea that this is componentized. However, remember that there are more than just one type of flash message. There are `message` from your app, but Laravel tends to use the paradigm of `success` and `error` as well. So you might want to check additional session keys and generate output based on them as well.

components/layout.blade.php - feedback for this:

- I know this is a work in progress, but I'd highly recommend importing javascript like Alpine and CSS like font awesome with Laravel Mix (or Vite) as soon as possible. Then, you will benefit on deploy from things like tree shaking and removal of css/js not in use.
- Generally, you want to define a section called `title` for your titles, and then can default back to a known one. For example:

```
<title>@yield('title', 'LaraGigs | Find Laravel...')</title>
```

This will display the default title when no title is set. Otherwise, on your other views - say like login or listings, you can do `@section('title', 'Log in')` and it will make that the title tag.

- again, use named routes when you can. Don't hard-code something like `/listings/manage` because you may want to change it later, too.
- quality IDE's should point out problems in your HTML code as well. For example, this line has duplicate `class` attributes, so the result is often non-deterministic: `<img class="w-24" src="{{asset('images/logo.png')}}" alt="" class="logo" />`
- Something simple - but when using a copyright year, you can use the current date from something like `Carbon::now()->format('Y')` or something even as simple as `date('Y')`

components/listing-card.blade.php Another benefit for using named routes is you can replace the `a` tag's `href` attribute with a simpler output. It may not be 'simpler' but it actually is. As it is, you have to know that the key and the bound values are `->id` on the `$listing` variable. Instead you could do this `href="{{ route('listings.show', ['listing' => $listing]) }}"` - while that may look longer, it actually is better. It's a named route, so you always know where you've implemented that data. You also don't have to know the key is `id` - under the hood, laravel calls `->getKey()` which allows you to abstract that. All you know is that it takes a listing. That way the url could be `/listings/28` now - but in the future it could be something like `/module-jobs/listings/private-view/28/show` or whatever - and you'd never have to change this blade file.

users/register.blade.php - a couple things:

- never **NEVER** output a password, even if authentication failed. Do not use `old('password')` ever. Those always need to be retyped.
- A lot of times the CSS class is added/changed on error. Remember, you can use something like `@class(['error-class' => $errors->has('email'), 'form-input')` for example. What that does is - depending on if there is an error with email, it will add the `error-class` - but it will always add the `form-input`. This directive takes care of rendering the proper classes and doing proper spacing, etc.

partials/_hero.blade.php - Do not use inline style. Instead, add the background image to your CSS and use an ID or class to target the element.

partials/_search.blade.php - Generally, you can repopulate the search term in search boxes. Most often, search is something like `q` or something predictable - but you could even pass it in as a variable to the include for the partial if need be.

listings/edit.blade.php - it looks like this is a lot of copy/paste from the create. You could make a partial for the form and include it in both places. Then, when you want to set the value, you can do this: `value="{{ old('title', $listing->title ?? null) }}"` which works like this: First, attempt to use the old submitted value - so when you have a validation error, it will put everything back so they can fix it. Next, if there is no old submission, render our default value. in cases where there is a `$listing` variable (that is, on edit), use the `title` attribute. In cases where there is no `$listing` variable (like create), null coalesce to the default of `old()` which is null.

listings/manage.blade.php - just as UX thing here: especially since there are no soft deletes on the Listing model, it's best to put some sort of dialog or confirm on the delete form. Otherwise people (or a cat) could click the button and that listing is gone forever.

Automated Testing

Automated testing is sometimes called unit testing. These names can get confusing because unit tests are a form of automated testing. Oh boy. But, let's just touch on this briefly.

Automated testing helps reduce bugs in new code and prevent regressions from new features and refactoring. In my projects, I've got an extensive test suite that allows me to upgrade packages with little fear. I know that I will run my tests and if they all still pass, the level of coverage over all the functionality I have will let me believe that there are no new bugs.

Unit tests are not a replacement for executing the code by hand, however. They should be used in tandem. Sometimes, automated tests are considered to be a form of documentation. When you name the tests appropriately, you can scan through the test suite and get an idea what all of the methods under test do and what you'd expect to see when they're exercised in certain ways.

There are a number of types of unit tests or automated tests - let me break them down into the main 3.

Unit test: the basic test - this is the smallest unit of code that needs to be tested. It can do one thing and should always have one output. For example, imagine a scenario where you have a model that has a method called fullName() - it builds the first name, a space, and the last name as the output. You may write a unit test that checks what happens when both fields are blank, what if just the first name is blank, what about just the second, what about if both are filled. There's no need to interact with any other part of the system and the logic is relatively straight forward to test. You can use PHPUnit in PHP, Chai/Mocha or Jest in JS.

Integration test: this is when you want to integrate at least one other block of code or functionality of your app. In some cases, this means testing how various classes interact with each other. In most cases, however, I take this to mean that it's different levels of the app: I want to test how my database interacts with this PHP code. But I'm not exercising a full end point of the application. For example, I might have a repository that retrieves only odd numbered streets of address objects. I may seed in a whole data set in my test into my database, then initialize the repository and call that retrieval method. I want to test that only odd streets are retrieved. That way when that code is used in a full implementation of the application, I know it's already been tested. Do not use the same database as you use for development as you might for integration tests. You can use PHPUnit in PHP, Chai/Mocha or Jest in JS.

End to end test: These are sometimes known also as feature tests. This is basically taking an expected visitor state, an expected user input, and processing it through an HTTP request, and measuring the expected output and system by products. For example, you might create an authenticated user, and send in a string that is 100k characters long to your blog entry end point, and expect and test that it returns a 422 error, a useful message about line lengths, and that no new database entries have been made. You can use PHPUnit in PHP, or Cypress or Selenium to exercise browsers.

End to end tests are generally where I focus when it comes to adding tests to a legacy system. Build the scaffolding around the functionality and then slowly get more detailed. Then, as you replace your functionality with new code, you make sure the tests still pass.

You might think then you should only create E2E tests - but that's not the case. Generally, they take much longer to run and set up. So, you'll find that a healthy mix of different types will likely cover your code base appropriately in reference to the time/expense required to create & run them.

Project Specific Testing Feedback

phpunit.xml - Here comes a lot of personal opinion. But, first, I'd say you should add the `stopOnFailure="true"` attribute to the `phpunit` element. This stops the tests on the first failure. My logic is such: if a test fails, you probably want to work on that - why have to hit ctrl-c to quit the tests? In addition, if one test fails, likely others will 'false fail' based on the reason for the first test failure. That is to say, if you forgot to do authentication correctly on an end point, why do you need to test all 5 ways of using that end point? The first failed, what do additional failed tests actually give you?

Second thing: use `null` as a queue connection. This means no queued jobs will run. Your code tests can check if events have fired and are wired up, but when you test that all the listeners have ran in queue, you're running far more than just the end point you're trying to test. You can get higher quality coverage by making your tests more targeted by launching listeners (after you've tested and verified they're wired up) manually.

Finally, there's a whole conversation on what feature vs unit tests are. This has been detailed later. I tend to use three: Feature, Integration and Unit. Your PHPUnit.xml file will have three sections should you adopt this practice. The most important thing is to have actual tests, no matter what you're doing.

tests/Feature/ExampleTest.php - I would remove this. This is just one more thing that has to run that could possibly fail some day. Remove code you don't need - especially test code. This helps in so many ways (code review, speed, IDE analysis, code analysis, etc). It also

reduces mental load. As you build out more complex projects, small things like these all add up.

Misc and Project Details

.styleci.yml - Mostly a personal opinion: Generally, if you're not going to use a particular service, you may want to remove the configuration. These can always be added later. Other developers assume you agree with, promote and have configured services that you retain configuration for. So, if it's not in use, remove it. Then, if you want to integrate later, you can always add it back / or generate your own configuration from the source of the integration.

Anyway, I prefer the tool `PHP_Codesniffer` for style stuff, and `Larastan` for code quality. While I think there are some options to run StyleCI locally, it's my opinion that the programmer should be responsible for applying style, not having that ran on a third party's platform committing to code you've already approved as your own.

README.md - a couple things here:

- first, I like the image that is used for the top of the readme. However, it doesn't have to be in the `public` folder - unless it's being used in other places on the internet. If it's just for this repo, which I assume it is, you can put it in a top level folder like `art` or something. It doesn't need to be available on the web.
- make sure to check spelling. You can enable a spell checker in your IDE usually. Normally those understand camelCase as well, so they'll let you know about spelling mistakes even in your variables. This example comes from the misspelled word `nessesary`. (which should be `necessary`)
- the instructions to add the credentials for MySQL to `.env.example` are wrong. Well, technically they could be right - but ... So the point is, don't add credentials into your example file unless they're very specifically not secrets. In this case, I can see there being confusion on whether you have to do that for production as well. We also know that people tend to re-use passwords, so I wouldn't suggest anyone put a password in the example file - unless they know what they're doing. A lot of times, in projects I work on, I will create a local database but I'll specifically point out that I've used a simple password like `password` - so it's obvious that I can 'share' that 'secret'

_laragigs_theme - I assume this was for generating the layout of the project before the programming started. In cases like this, I would not include this with the main project. As another programmer, I was confused and thought this applied to the system in some way. Repos are cheap/free, so create a different git repo and store this in there. Keep the design assets separate from code - and only import imagery and assets that you need.

Misc

This section is just miscellaneous things that don't necessarily apply to a single file.

- Definitely consider something like PHP_Codesniffer to validate and apply your code standards of choice. When you have multiple developers - or even for yourself - this can definitely help. A lot of the auto-formatting allows you to just write code, save it, and watch it transform into properly formatted code. This works automatically like 95% of the time. But you need to sometimes still apply the changes.
- Consider strict type declarations as you increase the complexity. You can do this with `declare(strict_types=1);` at the top of your files. This tells PHP not to type-juggle. You don't need to apply this now, and it should probably be done on a file-by-file pace, but it has saved me from a lot of bugs in the past. It also makes it 'slightly harder' if you're used to the loose-typing of PHP.
- Use ide-helper - this will generate all kinds of useful things for your IDE - including properties and methods on your models, facade auto complete, etc. You're struggling needlessly if you're not using this tool.
- When you have a license like `MIT` in your `composer.json` file, it's a good idea to also generate a `LICENSE` file in the root of your project. That will alert tools like Github what license your whole project is. In fact, you can create the file through their interface and it'll give you a number of templates - including the MIT license - to add.
- Things I might consider adding / doing better. I know this was a quick project and may be a work in progress:
  - Add policies. This determines the permissions that people have to do things. Then, you can add the checks as middleware or directly in the constructor of resourceful controllers.