Handling business logic in Laravel

The Essentials

What is business logic actually?

In programming, business logic is the part of a software program responsible for implementing the *business rules* that define how data should be created, modified, transformed, communicated and in other ways managed and controlled.

Business rules are the parts of the program (statements, functions, etc.) that are specific to what the application is about. If you are building a webshop for a Dutch company that sells Cheese for example, the business rules might define the sequence of steps in the checkout procedure, the ways that the shop wants offer discounts and special offers and algorithms that determine when a given piece of cheese is out of date.

In simple terms, business logic is all about how a particular organization, the one using the application, does its thing – it's their unique way of doing business.

Types of business logic

In a web application, business logic determines how the application's data is handled. An example where you can find business logic in a web application is input validation. Others (but not complete) are:

- **Data constraints** - the rules that limit the acceptable values for database table attributes, like its type, `nullable` or default value. This also includes referential integrity constraints for related data
- **Input validation** - the rules that determine which input values are acceptable for the business
- **Derived attributes** - data that behaves like a model's attribute, but is actually derived from other data. For example an invoice's total amount is derived as the sum of the amounts of its containing sold items
- **Workflows** - changes (additions, updates and deletions) that must be handled in a specific order, with specific checks and/or specific combination
- Attributes that are allowed to be **mass assignable**
- **Query scopes** - rules that determine ways how data should be fetched from the database. For example a specific sorting order
- **Casts** - rules that determine how attributes must be converted to common data types like DateTime or even encrypted

Design considerations: where to put business logic code?

It is the developers job to design these rulesets and algorithms to implement these business logic. For each algorithm there are three things you need to consider:

- The name of the function/method or variable that implements this business rule
- The actual algorithm or rule itself (the 'formula')

- The location (i.e. which class/file) you want to put it. This might be in
  - a View where the business rule is used
  - inside a Controller method, like you already did when implementing request validation
  - inside a Model.

As with all other (object oriented) programming design decisions, the proper choice is pending on:

- Responsibility driven design: "Who (i.e. which class) should be responsible for this?"
- Code duplication/reusability - It should be as easy to get a model's derived attribute as its normal attributes so it can be used in any view or controller that collaborates with that model. You shouldn't need to program the algorithm more than once.
- Maintainability - To prepare for future changes (business rules often change), you want to keep the amount of code files that hold the business rules as low and logical as possible. This makes it easier to locate and implement changes now and in the future.

Considering all this, the Model seems to be the correct place in most cases. A Model is just a class like any other class. So, it's easy to extend the functionality of that model with additional attributes and methods. And, of course, Laravel also provides features that help with this. The following section describes

Essential business logic in Laravel

Data constraints

These are business rules that define how the database should behave. Laravel has chosen to utilize migration files for specifying the database structure. Consequently, it is considered a best practice to define these business rules within the migration files by using column modifiers like `nullable` or `default` or foreign key constraints like `onUpdate`.

Derived attributes

In an ideal situation, derived attributes should behave exactly the same as normal attributes. When developing views, it should be perfect when the developer can't distinguish if in `{{ $user->name }}` the `name` is a normal or a derived attribute. Laravel provides a way to accomplish this: accessors. See if you want the Digging Deeper section. But you can get close by just defining a method like `getName` in the User model:

```
class User extends Authenticatable { // ... public function getName() {
return $this->first_name . " " . $this->last_name; } }
```

So it can be called in any view that handles a user like:

```
{{ $user->getName() }}
```

Workflows

Just like derived attributes, workflows (or at least the basic workflows, see Further Reading if you like) can be implemented as methods in a model.

In a workplace application for example you might have a `Job` model that holds information about a job. The business might require that when a job is closed, an `Invoice` needs to be created. This can be implemented by adding this to the `Job` model class:

```
public function close() { // Update the state of this job $this->state =
"Closed"; $this->save(); // Create an invoice based on this job
Invoice::create([ 'title' => $this->title, //... ]); }
```

Which can be called in for example a controller like so:

```
public function close(Job $job) { $job->close(); // ... }
```

Digging deeper

Accessors for derived attributes

In Laravel, accessors and mutators, often also called “getters and setters”, are used to manipulate and control the values of attributes (fields) in Eloquent models. They are defined as methods of a model that will determine the behavior of that attribute when it is set and/or read (get) using the following rules:

- The method name should correspond to the "camel case" representation of the true underlying model attribute / database column
- The method should return an `Illuminate\Database\Eloquent\Casts\Attribute` object that the defines the attributes get and set behavior

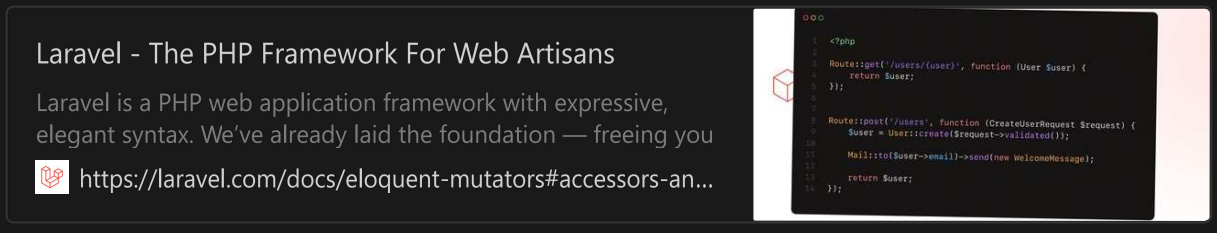
In the following example, we'll define an accessor for the `first_name` attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the `first_name` attribute. It returns an `Attribute` object which is created using its static `make` method that has one named argument: `get` that defines a callback function

```
class User extends Model { /** * Get the user's first_name attribute. */
protected function firstName(): Attribute { return Attribute::make( get:
fn (string $value) => ucfirst($value), ); } }
```

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may simply access the `first_name` attribute on a model instance:

```
use App\Models\User; $user = User::find(1); $firstName = $user-
>first_name;
```

This is basically a section of the following article in the Laravel documentation:



This mechanism can also be used to define derived attributes (Laravel calls this *value objects*). By defining an an accessor that does not correspond an existing attribute, Laravel simply adds the attribute behavior to the model.

For example, you can add an extra `name` (read-only) attribute, that is a concatenation of the user's first and last name like so:

```
class User extends Model { /** * Get the user's name attribute. */ protected function name(): Attribute { return Attribute::make( get: fn (string $value, array $attributes) => "$attributes['first_name'] $attributes['last_name']", ); } }
```

Then you may simply access the `name` attribute on a model instance:

```
use App\Models\User; $user = User::find(1); $firstName = $user->name;
```

Validation rules in Models

> ⚠️ **Note**: this is a bit of an opinionated design. Which means that it is not considered a best practice throughout the entire Laravel community but rather just the view of some specific people.

Instead of defining the validation rules (twice) in a controller, you can add something like this to a model:

```
class Faq extends Model { // ... /** * The rules for validating request and other input data. * * @var array|string[] */ public static array $validationRules = [ 'question' => 'required|min:10|max:1024', 'answer' => 'required|min:10' ]; }
```

And use it like this in for example a Controller:

```
public function store(Request $request): RedirectResponse { $validated = $request->validate(Faq::$validationRules); // ... }
```

In this way:

- Validation rules are only defined once
- This business logic is also located in an more appropriate location: it's corresponding model

Further reading

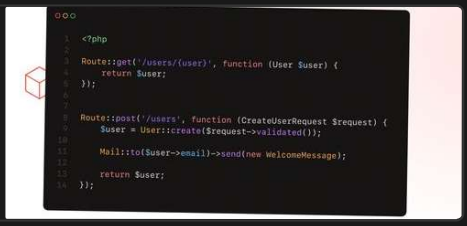
Mutators

There is a lot more to learn about accessors and mutators that we haven't explained, like Mutators. See the link:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent-mutators#accessors-an...



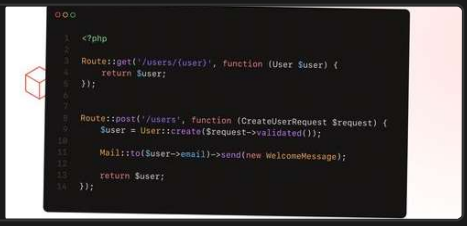
Query scopes

Query scopes are a way to define Model specific queries or even query parts like ordering. Especially with *Local Scopes* you can define and use model specific scopes. See:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent#query-scopes



Handling complex business logic

When business logic workflows becomes more complex, you might want to consider using the Repository or Service pattern:

Laravel | Repository & Service Pattern

Are you starting off your new Laravel project? Take a moment to plan ahead!

https://joe-wadsworth.medium.com/laravel-repository-serv...

For even more complex business logic workflows, there is also the Command pattern

Command Pattern: A different way to organize your S...

If you are like me, you probably create a Service layer where your application's business logic resides. And, if you are like most of

https://medium.com/devcupboard/command-pattern-a-ne...