

# Mastering Laravel Validation Rules

Go beyond the docs and  
choose the right rule every time.

**SAMPLE**  
MASTERINGLARAVEL.IO

# Table of Contents

|                                                                         |           |
|-------------------------------------------------------------------------|-----------|
| <b>Intro</b>                                                            | <b>5</b>  |
| <b>Chapter 1: Why Do We Validate?</b>                                   | <b>8</b>  |
| Security                                                                | 8         |
| Data Quality                                                            | 9         |
| Objections                                                              | 10        |
| <b>Chapter 2: Leveraging the Laravel Framework</b>                      | <b>13</b> |
| Where to Put Validation Logic                                           | 13        |
| Accessing Validated input                                               | 18        |
| How to Structure Validation Rules                                       | 20        |
| <b>Chapter 3: Beginner and Common Rules</b>                             | <b>24</b> |
| Required                                                                | 25        |
| Present / Present If / Present Unless / Present With / Present With All | 27        |
| Missing / Missing If / Missing Unless / Missing With / Missing With All | 33        |
| Nullable                                                                | 37        |
| Sometimes                                                               | 38        |
| Confirmed / Same                                                        | 40        |
| Boolean                                                                 | 44        |
| Integer / Numeric / Decimal                                             | 46        |
| String                                                                  | 52        |
| Alpha / Alpha Num / Alpha Dash                                          | 54        |
| ASCII                                                                   | 60        |
| Uppercase / Lowercase                                                   | 63        |
| Size / Min / Max / Greater than / Less than / Between                   | 67        |
| Date / Before / After / Equals                                          | 75        |
| JSON                                                                    | 82        |
| IP / IPV4 / IPV6                                                        | 85        |
| MAC Address                                                             | 89        |

|                                                                              |            |
|------------------------------------------------------------------------------|------------|
| ULID / UUID .....                                                            | 92         |
| URL .....                                                                    | 95         |
| File .....                                                                   | 98         |
| Image .....                                                                  | 100        |
| <b>Chapter 4: Intermediate and Less Common Rules .....</b>                   | <b>103</b> |
| Array .....                                                                  | 104        |
| List .....                                                                   | 116        |
| Distinct .....                                                               | 119        |
| In Array .....                                                               | 123        |
| Accepted / Declined .....                                                    | 127        |
| Bail .....                                                                   | 136        |
| Date Format .....                                                            | 139        |
| Different .....                                                              | 141        |
| Digits / Min Digits / Max Digits / Digits Between .....                      | 144        |
| Email .....                                                                  | 151        |
| Ends With / Starts With .....                                                | 154        |
| Exclude / Exclude If / Exclude Unless / Exclude With / Exclude Without ..... | 161        |
| Filled .....                                                                 | 166        |
| In / Not In .....                                                            | 168        |
| Enum .....                                                                   | 171        |
| Contains .....                                                               | 174        |
| Mimetypes / Mimes .....                                                      | 176        |
| Multiple Of .....                                                            | 179        |
| Password .....                                                               | 183        |
| Prohibited / Prohibited If / Prohibited Unless / Prohibits .....             | 186        |
| Regex / Not Regex .....                                                      | 193        |
| Conditionally Required .....                                                 | 197        |
| AnyOf .....                                                                  | 212        |
| Timezone .....                                                               | 215        |
| <b>Chapter 5: Advanced and Custom Rules .....</b>                            | <b>219</b> |
| Active URL .....                                                             | 220        |

|                                                       |            |
|-------------------------------------------------------|------------|
| Dimensions .....                                      | 223        |
| Exists .....                                          | 227        |
| Unique .....                                          | 231        |
| Current Password .....                                | 237        |
| Closure Validation Rules .....                        | 239        |
| Validation Rule Classes .....                         | 243        |
| Implicit Rules .....                                  | 249        |
| <b>Chapter 6: More Validation .....</b>               | <b>250</b> |
| Conditional Rules .....                               | 250        |
| Accessing Nested Array Data .....                     | 251        |
| The ConvertEmptyStringsToNull Middleware .....        | 252        |
| Request Class Dependency Injection .....              | 253        |
| Preparing and Sanitizing Data for Validation .....    | 255        |
| Additional Validation Steps Using Form Requests ..... | 259        |
| Validating Other Sources of Data .....                | 261        |
| <b>Chapter 7: Real World Examples .....</b>           | <b>266</b> |
| Validating an Address and Phone .....                 | 266        |
| Validating Transferring a Digital Asset .....         | 270        |
| Validating Conditional Date Requirements .....        | 274        |
| <b>End Notes .....</b>                                | <b>277</b> |

# Intro

Since my family doesn't celebrate Christmas, I'm spending Christmas Eve with my best friend's family. I'm learning about the Polish traditions of Wigilia. Having never eaten borscht before, I'm having quite an evening. To be honest, I don't like the soup. During the middle of my Oscar-worthy performance of "enjoying" this first course, I'm almost excited to hear my pager beep. (The pager gives you a good idea of what decade this was.) I excuse myself to call the number. My stomach churns and my heart races when I hear the voice of the executive chairman of the board at my company.

"Do you have any idea why the website might be down?"

My team isn't large; just me and a designer. And, I know that I've not planned any maintenance. I'm pretty certain my designer doesn't even know exactly how a website works, so it wasn't him. What's going on?

"Not that I'm aware of. What's happening?" Not my favorite question to ask the main boss.

"We planned a major fundraising drive for Christmas day, but we can't process any payments on the website right now. This needs to get fixed immediately!"

Now I'm wishing to be back feigning happiness over a bowl of borscht. This is going to be a long night. I excuse myself from the party and drive home to my computer.

I'll spare you the details of the long and tedious night of tracking down what went wrong. But suffice to say someone had hacked my server using a form I had not validated properly. They were able to create an account and inject software to try to steal customer credit card information. Luckily their attempt failed, but it also disabled our processor's system of accepting credit cards as well. We were quite lucky nothing major was wrong, but it was my fault we got hacked. I hadn't validated the form and someone took advantage of it.

This wasn't the only experience that awakened me to the value of user data validation. Throughout the years, I've experienced hacking attempts, legitimate customer mistakes, third-party data source changes, and more. Each of these problems could have been fixed with a solid validation scheme in place.

Whether you've been the victim of a hack attempt, or you've got a robust validation scheme in place, I wrote this book for you. If you're just getting started, I'll talk a bit about validation: the what, why and how. If you're an old-hand at this, join me as I dig into some of the more advanced concepts and reasons why to use specific validation techniques.

Even though Joel and I both participated in writing this book, it feels awkward to always use the plural "we". So from now on, I'll just refer to "you" directly and to myself as "I". In

addition to collaborating on the main content, Joel will also interject a few notes throughout the book. Those notes provide clarification, additional context or background information to help you get the most out of what you're reading.

In this book, I'll focus on real-world examples. You won't find any `foo`, `bar` and `baz` here. While it's not possible that I'll cover your exact situation, I'll aim to give examples that make sense in general. The Laravel documentation is the gold-standard of how to use all of the features of the framework. This book doesn't aim to replace the validation section, but to augment it and give some opinionated direction. The flexibility of Laravel can sometimes be daunting, so I'll focus on tried-and-true suggestions.

This book is split up into several chapters to help keep everything organized. You can use it as a reference, but I recommend starting from the beginning if you're new to validation.

Chapter 1 covers the reasons why validation is necessary. It's not just security! I'll also cover data integrity and quality issues. Then, I'll tackle a couple common objections you might have heard against writing thorough validation logic.

Chapter 2 describes the ways that validation logic can be organized and written in Laravel. I'm not one to shy away from giving my opinion, so you'll get my recommendations as well as why I feel that way.

Chapter 3 focuses on beginner, or the most common, validation rules. These are the rules you're likely to use all of the time and that are relatively simple to understand and implement.

Chapter 4 dives into the intermediate and less common validation rules. These rules are still important even though you may run into them less often, or they may require a bit more understanding to use correctly.

Chapter 5 details more advanced and custom validation rules. The rules in this section require a bit more explanation.

Chapter 6 is a mix of things that are important, but don't fit into any other chapter. You'll find a few more features of the validation toolset, discussion regarding the initial setup of Laravel projects, and ways to use validation you might not have thought of before.

Chapter 7 contains three real-world examples. In each example, I'll start by describing the application scenario. Then, I'll show the code from our `FormRequest` class. Finally, I'll detail the reasons for each validation rule and what could happen if it didn't exist.

In order to keep the size of this book manageable, I won't be focusing on a few parts of the Laravel validation toolset. These include error message customization, Blade and user-facing error display, and form repopulation. While these are important parts of the user experience, I'm confident you can handle these without our guidance.

We're happy you're here! Congratulations on taking the next step toward being the best Laravel programmer you can be. Let's get on to the validation!



# Chapter 1: Why Do We Validate?

When you need to hit a tight deadline, it can be easy to dismiss the value of validation. It takes more time, it requires you to think way beyond your immediate requirements, and the tests can be quite involved.

If you're hacking together your new project idea - you know the one, that will change the world - you might think you don't have time for it. Or, because you know your user-base, you trust they will be well-behaved. Why would anyone ever type in a string of five thousand A's into a city field? Your users are better than that; they've got more important things to do than to try to break your product.

Or perhaps you are integrating with a large third-party data provider and they have a thoroughly-documented data model. There's no way that they'd change their data structures without telling you in advance. They have a great communication strategy and multiple layers of testing and QA in place. You can trust them.

Because of these reasons and assumptions, many projects tend to leave validation until the end. It becomes an after-thought. You might think "we don't need to spend a lot of time on validation, we need to get the features out the door!" Or, "we're not a target." Or, "I'll get back to that later." All these objections will be addressed later in this chapter.

What's arguably worse than having no validation at all is having validation that's incomplete. Someone started writing very basic validation rules, but never took the time to complete them in more detail. Once this is done, it's easy to keep overlooking it. It's often assumed it was probably done correctly and completely, right? Why would I go and examine that if it's already done and in the codebase?

Validation is important, and shouldn't be left until the end. Since you're reading this book, we hope you already believe that, but let's just make sure we all are on the same page. (Book pun!)

## Security

That will never happen to us. Why would anyone want to hack our website?

"That won't happen to us" is one of the most common arguments we've heard to ignore validation. Developers or stake-holders will agree that web apps are hacked, but they'll say that they're not a juicy target. They might feel that since there are so many other targets

available, the chances are so low that they'll get "picked."

Let's dismantle this.

A lot of security breaches are using automated tools. Nefarious actors set up a tool to scan the internet for hosts and software that match a specific pattern. Then, the automated tool throws tons of potential attacks against the site or host until one is found to work. Arguing that you're not a target could be correct, to an extent. You're not a target specifically, but everyone on the internet is a target in the aggregate. Hacking attempts no longer take the attention of an individual focusing on a specific site to be successful.

How does this really work? We're not going to explain it step by step, but here's the gist of how this happens. A host is identified automatically from a network scan on IP addresses for a specific set of open ports, from a search engine like Shodan, or from the Certificate Transparency project. Then, tools like Metasploit are launched with pre-defined lists of exploits that are attempted one after another against that target. When a vulnerability is found, tools like BeEF are used to inject Javascript to remotely control browsers. The end result is an attacker now adds another "anonymous" hacked browser, host or software app to their collection. The hacker might not even know what vulnerable system gave them control of this resource.

It's true that some websites and apps are targeted on purpose, but there is also value in a large collection of faceless or anonymous websites and apps that have been hacked. It's not about you specifically, it's about collecting a large network of sites to do things like distributed denial of service attacks. This is when your site, software or server is used - likely without your knowledge - to launch a distributed attack on another internet user or site. Not only will you have to pay for things like bandwidth overages, but you may also be accused of participating in the attack.

Also, it's not great to have someone in your server and code base that you didn't authorize - like a random visitor to your house that you don't know. Maybe they aren't there to attack you, but you'd rather they weren't there at all.

## Data Quality

I need a report of this data - we need to start doing data analysis.

When you type your phone number, what is the format that you choose? Do you use a combination of parenthesis and dashes? Do you get clever and use periods? Do you put a

space? What about a country code? Does that include a plus?

What about postal codes?

If you have a status column in your database, you wouldn't allow a mix of `1`, `0`, `true`, `false`, and `active` would you? So why would you allow that in other fields like a phone number?

Whether you're going to be doing reporting or just displaying data, uniformity is important. This is even more important when you have to transfer your data from one system to another, or convert from one data storage mechanism to another.

There are two ways to obtain data uniformity. The first and most seamless for the user is to accept any data and then automatically convert it into the format the system accepts. This works for easy things like status toggles and pre-defined lists of things, but when it comes to free-form data, trying to automatically normalize it can be quite a headache.

That's why we focus on the second way to do this: validation. If you reject data that's not in the proper format, you shift the formatting to the user. While this is slightly more work for the user, a human is much more capable of manipulating their data into the desired format than expecting the application to clean it up automatically.

## Objections

Even when programmers get on board with validation, we still hear a number of common objections. Let's talk about those.

My front-end dev is a validation master! I don't need to do anything on my end.

The near-instant feedback of front-end validation compared to the server round trip provides a great user experience. You can't rely on this for your security and data quality, however. Why not?

If you've ever used an API, you know that you can post to endpoints without having to use the browser. It's quite possible that a villainous visitor may reverse engineer your requests and take a stab at posting corrupt data directly to your backend application. All of that beautiful validation done in the front end can't protect against this.

Front-end validation can also fail. When Javascript has an error, if it's not handled properly, it can stop the rest of the code from executing. So, it's quite possible that a bug in another piece of code could disable all of the front-end validation. We're also relying on the browser vendor to implement the Javascript properly and for the user to have a fully functioning,

current version of the front-end code on their device.

Some validation just can't happen in the front-end without exposing too much sensitive data to the public. For example, if you want to validate someone's secret government identification number, you wouldn't want to send that to the browser for anyone to view. That sensitive information is best kept on the server and validated under your own control and with strict privacy.

Because of these reasons, it's best to implement strong validation on the server side. We're not saying you shouldn't validate on the front-end, but that the validation system should be a coupling of the two layers. The best scenario for a user is that their data is validated on the front-end and it smoothly passes your back-end checks without them ever knowing it was being re-validated.

I don't have time to write all of this validation!

It's common to hear people complain when they're learning to type. If you'd just let them hunt-and-peck with one finger on both hands, they'd be done by now. This typing is uncomfortable and taking forever! Soon, however, they learn to type and become much faster. After learning the skillset, there's no way they could go back - it's better and faster this way.

The same thing is true with validation. If you're not used to doing it, it can seem like it takes a long time. However, once you develop the skills and make it a habit, it doesn't take as long as you might think. Plus, Laravel gives us tons of built-in tools that make it much easier than writing validation all by hand.

True, it does take a little bit longer than not writing it at all. But the time invested is dwarfed by the value you get from writing it.

**Joel says...** This is a perfect spot to mention the importance of practicing the techniques presented in this book. Reading chapter after chapter is a good start, but it's also important to actually try the code and apply it in a real-world project. Taking the time to practice will help you internalize these skills, making you like that proficient touch typist on future projects.

My users are trusted because they're internal.

When programmers create internal applications, a common objection against investing in validation has to do with the trust of the team members. If you have permission to do anything as an admin, or if you work for us, we don't need to validate your data. You won't be doing anything bad like those faceless hackers on the internet.

But imagine a scenario where your user has just spilled hot chocolate on their keyboard. That sticky mess has caused their **a** key to get stuck down. Before they know it, the **a** has filled in 10,000 times into your form field and the form is submitted. Ok, so maybe that specific scenario is a stretch... but we've also seen quite a lot of "funny" videos with cats walking on keyboards. Sometimes mistakes happen. Validation can prevent them.

**Joel says...** Validation can also help prevent misunderstandings where a user expects something to work differently than the development team intended. For example, maybe they have two accounts for the app and think they should be able to "merge" them by updating them both to use the same email address. On the other hand, the developers built the application on the assumption that every user account has a unique email. Oops! Validation can catch misaligned expectations like this as well.

I can't catch every validation issue, so I don't really try.

That's right, it's really tough to catch "every" validation issue, but that's not a excuse to give up on validation completely. You might not be able to escape every car accident on the road, but you still wear your seat belt, right?

We're not saying you should spend all your nights worrying about whether you've done the proper validation. Instead, our goal with this book is to help you get a reasonably good baseline in place and have confidence your app works as you intended.



Some pages are not included in this sample.

## Required

The **required** rule is a simple but powerful rule. It defines that this field is required to be in the request input and that its value must not be "empty" similar to how PHP defines empty. Empty values include a string with no contents, a **null** value, an array or countable object with no entries or an undefined upload file. A numeric value of **0** is **not** considered "empty" according to Laravel's validation, however.

**Use:** Requires values to be available in the request. You may need to reject a request if key information is missing.

### Success

```
$rules = [
    'name' => 'required',
];

// The name is a string value
$incoming = [
    'name' => 'Wanda',
];

// The name is an array with contents
$incoming = [
    'name' => [
        'first' => 'George',
        'last' => 'Washington',
    ],
];

// The name is a numeric value
$incoming = [
    'name' => 1.23,
];

// 0 as a name satisfies the required rule
$incoming = [
    'name' => 0,
];
```

## Failure

```
$rules = [  
    'name' => 'required',  
];  
  
// The key `name` is not defined  
$incoming = [];  
  
// The name is null  
$incoming = [  
    'name' => null,  
];  
  
// The name is a blank string  
$incoming = [  
    'name' => '',  
];  
  
// The name is an array with no contents  
$incoming = [  
    'name' => [],  
];
```

**Joel says...** If you want to dig even deeper in understanding **why** a particular validation rule works the way it does, take a look at the Laravel source code. The main place to look is inside of the `vendor/laravel/framework/src/Illuminate/Validation` folder. For example, to see exactly how the `required` rule works, look at the `validateRequired()` method inside the `Illuminate\Validation\Concerns\ValidatesAttributes` class.



Some pages are not included in this sample.

## Email

The `email` validation rule enforces email address validation on the value. The rule combines a third-party package `egulias/email-validator` and the php method `filter_var()` with the `FILTER_VALIDATE_EMAIL` parameter into a single interface. You can choose the type of validation you want by passing a parameter to the `email` rule. If you do not specify one, by default it uses the `rfc` configuration. You can combine options to have a more thorough validation if need be.

The options you can choose are as follows:

- `rfc` which validates the email based on RFCs 5321-22, 6530-32 and 1035. RFC validation is split between warnings and errors. This one only fails on errors.
- `strict` which validates the email as above, but fails on warnings as well.
- `dns` checks DNS records at the domain of the email to indicate if that domain processes emails. It does not validate the format of the email or that the email exists.
- `spooft` which validates for multiple UTF-8 characters that can be used to "spooft" a domain by tricking a reader with similar characters, like the Cyrillic letter `р` which is visually identical to the English `p`.
- `filter` which runs the `filter_var` internal PHP validation on the email address format.

There are a couple of things to consider when using or stacking these parameter configurations. First, when using the `dns` check, the validation requires your server connect to and query DNS records. This may slow down your validation significantly. In some situations, you may not have access to run DNS queries. Also, when using the `spooft` option, note that it does not guarantee error-proof email. It just validates using the `intl` extensions `SpooftChecker` class which can help detect characters that may be visually confusing. If you are not making email addresses available for other visitors to interact with, this is probably not necessary.

It's important to note that validating an email address against something like the `rfc` configuration may not guarantee an email address that you expect. Since it allows warnings, email addresses like `@address.com` would pass the validation. (A warning is issued because of the spaces, but this is not the strict implementation.)

It's been our experience that the average programmer doesn't necessarily know all of the rules for a valid email address. Don't be alarmed if you see something like `some|thing@here.com` as a validated email address as it does pass the RFC specifications.

**Use:** Validation of email addresses of various levels of strictness or correctness. Useful to help confirm deliverability of emails and help reduce user typos.

## Success

```
$rules = [  
    'contact_email' => 'email',  
];  
  
// this email address passes the RFC (Default) configuration  
$incoming = [  
    'contact_email' => 'email@address.com',  
];  
  
// empty string value skips (passes) validation  
$incoming = [  
    'contact_email' => '',  
];
```

## Success

```
$rules = [  
    'contact_email' => 'email:strict,dns',  
];  
  
// this email address passes the RFC validation  
// with no errors or warnings.  
// in addition, the gmail.com returns MX DNS records indicating  
// email deliverability  
$incoming = [  
    'contact_email' => 'email@gmail.com',  
];
```

## Failure

```
$rules = [  
    'contact_email' => 'email',  
];  
  
// this fails the RFC standard configuration as it's missing a user  
$incoming = [  
    'contact_email' => '@address.com',  
];  
  
// this fails the RFC configuration because of the special character  
$incoming = [  
    'contact_email' => '<special@character.com',  
];  
  
// null values fail validation  
$incoming = [  
    'contact_email' => null,  
];
```

## Failure

```
$rules = [  
    'contact_email' => 'email:dns',  
];  
  
// this email address has a domain that does not resolve  
// and otherwise does not return the proper DNS records  
$incoming = [  
    'contact_email' => 'email@i-do-not-accept-email.com',  
];
```

**Joel says...** Email validation is useful to prevent certain types of errors, but if you really need to make sure someone has provided their valid email, the best method is to also use Laravel's email verification feature which sends the user a message containing a link they must click.



Some pages are not included in this sample.

# Chapter 7: Real World Examples

Learning the fundamentals of validation and the rules available to you in Laravel is important. But, sometimes it can be a challenge to take that knowledge and convert it into real life. In this chapter, I'm going to give some examples of validation similar to requirements you may have.

First, I'll describe the scenario. This will include what we're probably doing with this application from a high-level, and then what this particular request is responsible for.

Next, I'll share the form request. While there are a few ways to run validation, I mainly recommend using form request classes. This is especially useful because we can group together pre and post validation functionality into one place.

Finally, I'll discuss why each of these rules exist. I'll talk about how these rules protect our data integrity and ensure security. In some cases, there may be alternative ways of validating or cleaning the data which I'll mention.

Let's get started.

## Validating an Address and Phone

In this example, we need to validate a US-based address. We're going to assume we have the email address for the current user, but we may want to contact them via telephone. Let's review the rules we know about US addresses and phone numbers (or at least the rules we're going to honor in our example).

- Address must have at least one street address line, they may have two
- Address must have a city
- Address must have a state abbreviation
- Address must have a 5 digit postal code, may use an extended postal code separated with a dash
- Phone may have a 10 digits, may use a phone extension number

From a data perspective, we have a couple things that we need to be concerned about. First, each one of our MySQL database columns has a maximum length. Our address fields have been defined as `VARCHAR(255)`. The city field is `VARCHAR(64)`. The state field is `CHAR(2)`. The zip, phone and phone extension are all `VARCHAR(16)`.

Below is the form request class we will use for validation.

```
<?php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class AddressRequest extends FormRequest
{
    public function rules(): array
    {
        return [
            'address' => [
                'required',
                'string',
                'max:255',
            ],

            'address2' => [
                'nullable',
                'string',
                'max:255',
            ],

            'city' => [
                'required',
                'string',
                'max:64',
            ],

            'state' => [
                'required',
                Rule::in(array_keys($this->getStates())),
            ],

            'zip' => [
                'required',
                'regex:/^\d{5}(-\d{4})?$/ ',
            ],

            'phone' => [
                'nullable',
                'required_with:phone_ext',
                'digits:10',
            ],
        ];
    }
}
```

# Time for the Full Book!

Interested? Probably. I would be. Your next step is to [click here](#).

- Seven chapters, from your first **required** rule to custom rule classes
- Every rule explained, with examples of input and validation results
- Opinionated advice on where validation logic belongs and how to structure it
- Real-world examples with rationale

---

**Get the full book at**  
**masteringlaravel.io**

---