Laravel Validation

Validation is the process of checking the incoming data. By default,
laravel provides the base controller class that uses the ValidatesRequests trait to validate all the incoming Http requests.

Let’s understand the validation through an example.

We will create an application in which we add the name of the student.

Data Validation Made Easy in Laravel

Laravel provides several ready-to-use validation rules for when your application’s users submit data via forms. You can mark input fields as required, set a minimum or maximum length, and require unique (non-duplicate) entries and valid email addresses. The Laravel validator checks if the input satisfies these rules or any others you specify.

These Laravel validation rules include:

required — The field data must not be null or empty.
array — The field data must be a PHP array.
bail — The validation rule stops executing after it encounters its first validation failure.
email — The field data must be a valid email address.
unique — The field data must not have duplicates in the database table.

All validation methods have pros and cons, but their variety enables you to choose the best approach for your needs. Depending on your chosen method, Laravel validation can occur in several ways, with manual or automatic error messages.

The most common method is code>validate, used for incoming HTTP requests. This method is chained to the request data, executing the validation rules. You can separate the rules for each field with commas, as seen in the example below.

```
use Illuminate\Http\Request;

public function store (Request $request){
  $validated = $request->validate([
        'email' => ['required, unique:users, email, bail'],
        'name' => ['required'],
    ]);

}
```

Here, email is a required input, meaning it can’t be null. Additionally, it must be unique in the users database table, ensuring the same email address isn’t registered twice. The last rule dictates that the email address must also be valid. Otherwise, the validation process ceases. The name field is required but has no other rules.

Validation Basics

To better understand validation methods, consider the following example. You’ll define a route for the end point and create a controller to validate and process the request data.

First, create a simple endpoint that allows users to store their emails and passwords.

Define the Route

Laravel routes are defined in the routes/web.php file for a web application or routes/api.php for an API. Fo

r this example, use api.php:

```
use App\Http\Controllers\UserController;

Route::post('/store', [UserController::class]);
```

Create the Controller
Run this Artisan command to create the controller:

```
php artisan make:controller
```

UserController
This command creates a UserController.php file in the app/Http/Controllers directory.

Now, define a store method to validate data entering the store endpoint before storing it.

This example will validate the following fields using these rules:

email — Should be unique, a valid email, and must be required
password — Should have a minimum length, password confirmation, and must be required
age — Must be a number and must be required

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UserController extends Controller
{
    /**
     * Store new user details.
     *
     */
    public function store(Request $request){
        $validated = $request->validate([
            'email' => 'required|unique:users|email',
            'age' => 'required|numeric',
            'password' => 'required|min:7|confirmed'
        ]);
        // After user data is validated, logic to store the data
    }

}
```

The confirmed rule allows you to require a particular field twice to verify that the data is accurate, such as users re-entering their passwords during registration.
This rule requires a field called password_confirmation, whose data must match the password field.

Display Error Messages
If the validation criteria are met, your code will continue running normally. If validation fails, an IlluminateV alidationValidationException exception is thrown, and the appropriate error response is returned.

The example is based on an API, which returns a 422 Unprocessable Entity HTTP response in JSON for mat. For web applications, it would redirect to the previous URL to display the error message, and the req uest data flashed to the session.

You can use the $errors variable in your views to display returned errors:

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Advanced Validation
Laravel provides another method of writing validations called form requests. A form request is a custom re quest class that organizes validations and declutters your controller.

They have their own validation and authorization logic suitable for large input volumes and can be used to define validation rules and customize error messages.

To create a form request, run this Artisan command:

php artisan make:request StoreUserRequest
This command creates a StoreUserRequest.php file in the app/Http/Requests directory and contains two default methods:

rules returns validation rules for request data.
authorize returns a boolean to denote whether that user has permission to perform the requested action.
Convert the previous example to use a form request.

```
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        // Add logic to check if the user is authorized to submit this data.
        return true;
    }
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'email' => 'required|unique:users|email',
```

```
            'age' => 'required|numeric',
            'password' => 'required|min:7|confirmed'
        ];
    }

}
```

To customize the error messages of these rules, you may override the messages method in the FormRequest class.

```
/**

     * Get the error messages for the defined validation rules.
     *
     * @return array
     */
    public function messages()
    {
        return [
            'email.required' => 'An email address is required',
            'email.email' => 'The email address must be valid',
            'password.confirmed'=>'Re-type your password as password_confirmation, passwords does not match'
        ];

    }
```

Note: The data name and validation rule are separated by a period (.) before the message data.

Custom Validation

To create custom validation, you can use a Validator facade instead of validate. The validator instance contains two arguments: the data to be validated and an array of validation rules. These two arguments are passed to the ::make method on the validator facade, generating a new validator instance.

```
use Illuminate\Http\Request;

public function store (Request $request){
        $validator = Validator::make($request->all(),[
            'email' => 'required|unique:users|email',
            'age' => 'required|numeric',
            'password' => 'required|min:7|confirmed'
        ]);
        if ($validator->fails()) {
            // Return errors or redirect back with errors
            return $validator->errors();
        }

        // Retrieve the validated input...
        $validated = $validator->validated();
        // Continue logic to store the data

    }
```

If you want to add an automatic direct, you can execute the validate method on a preexisting validator instance. If validation fails, an XHR request produces a JSON response with 422 Unprocessable Entity as the status code, or the user will be redirected immediately.

```
$validator = Validator::make($request->all(),[
'email' => 'required|unique:users|email',
'password' => 'required|min:7|confirmed'
])->validate();
```
You can also customize your error messages by passing a third argument called messages to Validate::make method:

```
$validator = Validator::make($request->all(),[
            'email' => 'required|unique:users|email',
            'age' => 'required|numeric',
            'password' => 'required|min:7|confirmed'
        ], $messages = [
            'required' => 'The :attribute field is required.',
]);
```

Summary

Performing data validation is crucial for keeping your dataset clean, correct, and complete. Data validation allows you to eliminate errors in your data that could potentially corrupt or otherwise impact your project. Validation becomes increasingly important when working at scale and with large amounts of data.

Laravel enables numerous flexible approaches to ensure the integrity and accuracy of the data that passes through your application. You can achieve complicated validation logic with default and customizable methods, making your codebase well-structured and more easily reusable.