Form request validation

The essentials

In the fast-paced realm of web applications, validation serves as the indispensable guardian at the gates of data integrity. Like a skilled sentry, its primary mission is to fend off chaos by ensuring that only the right data marches triumphantly through the gates, thus preventing the missteps and mishaps that can derail user experiences. Without robust validation, your application risks becoming a wild frontier where errors roam freely, wreaking havoc on functionality and user trust. It's not just about thwarting user errors; it's about sculpting an environment where accuracy reigns supreme.

There are two main reasons why validation is essential to any application:

- **Usability** - Nielsen's 4th heuristic - Preventing User Errors - dictates that you need to design your system such that it is impossible for users to create errors. Validation can prevent the application from processing form input data that is not valid. For instance when the user submits an empty form.
- **Security** - From security perspective, it is a ground rule to never trust your users. This is crucial because entrusting users with unverified data input opens the floodgates to potential security vulnerabilities you might never heard of.

Server and Client Side Validation

It is important to know that there are three ways of validating user inputs. Each way has its own advantages and drawbacks. For this course (and from security perspective) there is only one that is really essential. The rest is more of a nice to have. Let's list them first:

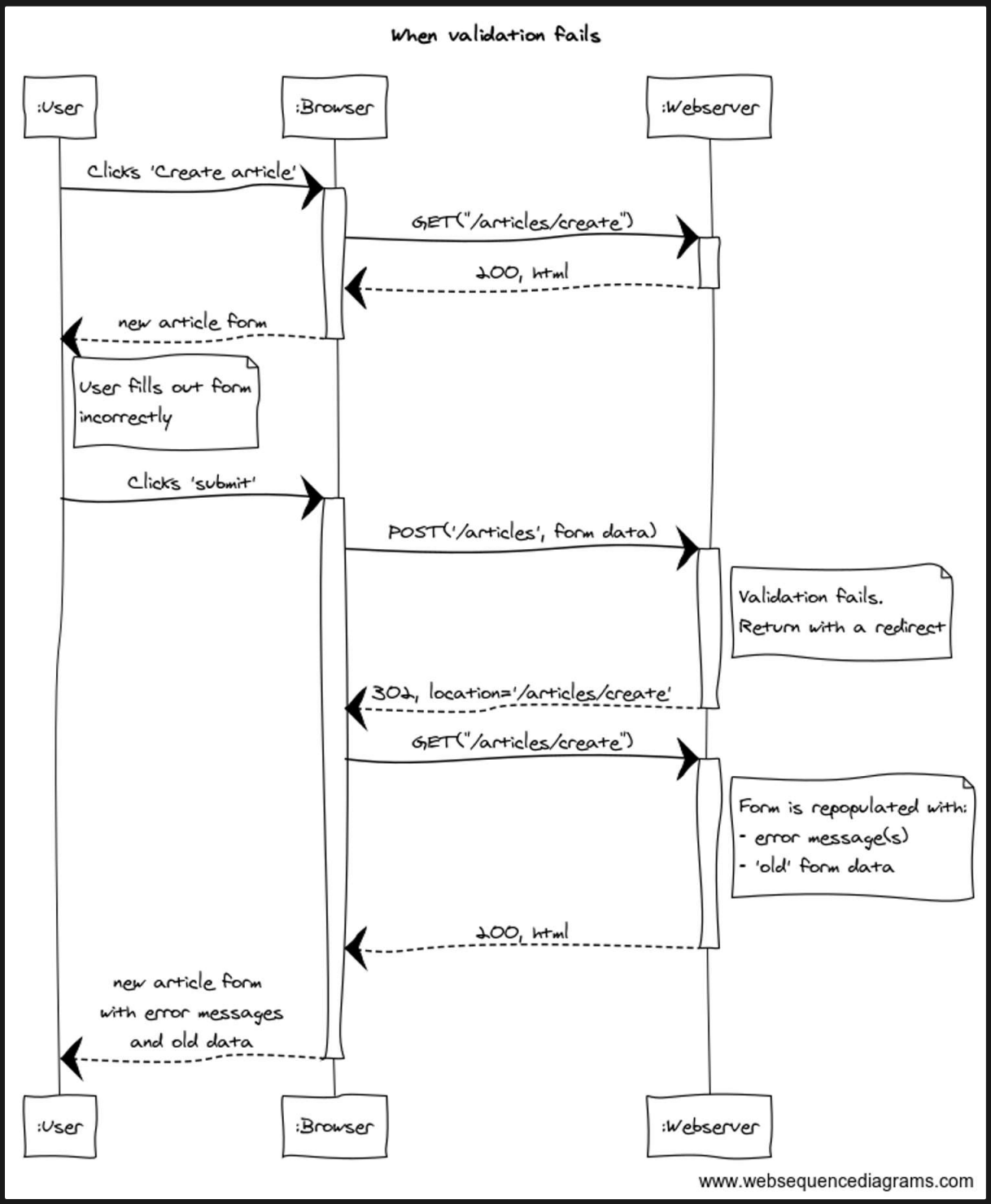
- **Server side validation** - The form will always submit and the submitted data is validated before it is processed.
- **Client side HTML validation** - The form elements are enhanced with additional attributes that mandate matching input values prior to form submission.
- **Client side JavaScript validation** - It's like HTML validation, but here the page has JavaScript code that checks the form input values

Each approach boasts unique advantages and limitations. Client-side JavaScript excels in usability, enabling validation in real-time as the user inputs data, a capability that eludes server-side validation due to the necessity of form submission. However, from a security standpoint, server-side validation stands as the sole viable method, as it prevents potential exploitation by hackers who can easily bypass client-side measures. HTML validation, resembling JavaScript in some aspects, is simpler to grasp for developers but falls short on the usability front as it offers limited control over presenting validation results to users.

> ! Remember, **server-side validation** is an absolute necessity, primarily for security purposes, and while it complements usability to some extent, it stands as the sole indispensable form of validation in this course.

Introduction to Server Side Validation

The entire process can best be described by the following sequence diagram:



In this diagram you can see that:

1. When the user wants to create a new article, he navigates to the required form page.
2. The browser sends the GET request, and the server responds with the form, which is presented to the user.
3. The user fills out the form, and (accidentally) makes a mistake.
4. When he clicks the submit button, the form data is sent to the server via a POST request.
5. The server checks the form data, and sees that something is not according to the validation rules. It responds by storing the error messages and old data in a temporary storage called a 'session'. A process which is called 'session flashing'. Then it returns a redirect to the same form
6. The browser automatically responds by sending a GET request to the given location
7. The server then responds by rendering the same form. Only this time, he uses the flashed data to render feedback information about the user error(s) and repopulating the form with the data the user previously filled in.
8. This is returned and presented to the user so he can fix his mistake and resubmit

Laravel Server Side Validation

Laravel's basic features provide support for server side validation in various ways. In this chapter we'll introduce the one approach that supports the best of what is discussed so far. When dealing with (server side) validation in a Laravel app, a developer needs to address the following three aspects:

- Validating incoming requests, by checking the input against a given set of rules
- Rendering the error messages in the redirected view when validation fails
- Repopulating form inputs with old values in the redirected view when validation fails

Validating incoming form requests

To validate an incoming request, the request's `validate` method can be used, for instance like this:

```
$validated = request()->validate([ 'title' => 'required', 'body' =>
'required', ]);
```

The example uses the `request()` helper to get the current request object. The parameter of `validate` is an associative array where each key is the key of a request data item where the associated rule must be validated against. The value is the actual rule.

The `validate` method call triggers the validation process. When validation fails, Laravel will stop executing the rest of the handler method, and returns the redirect response, which redirects the user back to the previous page. When validation is successful, the `$validated` variable contains an associative array that contains all the validated keys and values and Laravel will continue executing the rest of the handler method.

⚠️ Be aware that `required` is the only rule that is mandatory for this course. Laravel however supports a lot of other rules you might want to use. See the Digging Deeper and Further Reading sections if you want to know more about this.

Rendering error messages

When validation fails, Laravel also flashes the error information to the session and makes it available in the redirected view via the `$errors` variable. You can use different ways to use this variable to render its content, for instance:

```
<!-- /resources/views/post/create.blade.php --> <h1>Create Post</h1> @if ($errors->any()) <div class="alert alert-danger"> <ul> @foreach ($errors->all() as $error) <li>{{ $error }}</li> @endforeach </ul> </div> @endif <!-- Create Post Form -->
```

Repopulating form elements

Laravel supports repopulating form inputs via the `old()` helper, for example:

```
<input type="text" name="title" value="{{ old('title') }}">
```

In this example if no old input exists for the given field, `null` will be returned. If there must be some other value returned, for example in an edit scenario, you can set the default value like this:

```
<input type="text" name="title" value="{{ old('title', $post->title) }}">
```

Digging deeper

Advanced incoming request validation

As already mentioned, Laravel provides several methods for validation. Especially the following combination of alternatives provides a clean way of doing the basic CRUD tasks.

Using the injected request variable

Another way to access the current request object in Laravel is by injecting it as a parameter in the route handler method (see: **Routing - Dependency injection**). This is actually the preferred way by Laravel since it is used when it's generating a resource controller for you. For example s `store` method in a controller:

```
public function store(Request $request) { $validated = $request->validate([ 'title' => 'required', 'body' => 'required', ]); // ... });
```

Using the `$validated` variable

As already mentioned, the `validate` method returns an array that contains all the validated request attributes keys and values. The reason why Laravel does this is so it can be used in a mass assignment statement for creating or updating Eloquent models (see: **Eloquent ORM - Mass assignment**), which makes controller methods really clean. For example:

```
public function store(Request $request) { $validated = $request->validate([ 'title' => 'required', 'body' => 'required', ]); Post::create($validated); return redirect('/posts'); });
```

Other validation rules worth mentioning

The following rules are good to know if you want to dig a little deeper in this. Besides this, it is also good to know that Laravel allows you to combine multiple rules for an attribute.

Optional attributes

When an attribute is optional, the `nullable` rule is used:

```
$validated = request()->validate([ 'title' => 'required', 'tags' => 'nullable' ]);
```

Numeric fields

To check whether an attribute is numeric, the `numeric` rule can be used:

```
$validated = request()->validate([ 'title' => 'required', 'rating' => 'numeric' ]);
```

Combining rules

You can combine multiple rules for an attribute by passing an array:

```
$validated = request()->validate([ 'title' => 'required', 'rating' => ['required', 'numeric'] ]);
```

Adding sizes

You can use the `min:` and `max:` rules to specify a minimum and/or maximum length or value:

```
$validated = request()->validate([ 'title' => ['required', 'min:12'], // title must be at least 12 characters 'rating' => ['required', 'numeric', 'min:0', 'max:255'] // value between 0 and 255 ]);
```

Advanced error messages: the Input Feedback UI pattern

The Input Feedback UI pattern offers a more user-friendly approach of providing feedback when validation fails. See:

> Input Feedback design pattern
>
> Design Pattern: The user has entered data into the system and expects to receive feedback on the result of that submission.
>
> https://ui-patterns.com/patterns/InputFeedback

To summarize this, you must explain that:

- **An error has occurred** - by displaying a box at the top of the page (so that the user does not need to scroll the page to find out that an error occurred), preferably colored red to signal an error.
- **Where the error occurred** - by highlighting the fields (by changing their colors) that caused the
  error (using CSS).
- **How the error can be repaired** - by providing information on what needs to be different in order for the field to validate directly next to the field causing the error.

Laravel provides some convenient directives to implement this. You may use the `@error` Blade directive to quickly determine if validation error messages exist for a given attribute. For example adding a class when a error message for the `name` attribute is present:

```
<input class="input @error('name') is-danger @enderror">
```

Or render an exclamation icon next to an input:

```
@error('name') <span class="icon is-small is-right"> <i class="fas fa-exclamation-triangle"></i> </span> @enderror
```

Within an `@error` directive, you may echo the `$message` variable to display the error message that belongs to that attribute:

```
@error('name') <p class="help is-danger">{{ $message }}</p> @enderror
```

Combining this into a complete form input with label, placeholder, validation error messages and repopulating (using Bulma) might look like:

```
<div class="field"> <label for="name" class="label">{{ __('Name') }}
</label> <div class="control has-icons-left has-icons-right"> <input
type="text" name="name" placeholder="{{ __('The name of the city') }}..."
class="input @error('name') is-danger @enderror" value="{{ old('name') }}"
autocomplete="name" autofocus> <span class="icon is-small is-left"> <i
class="fa-solid fa-city"></i> </span> @error('name') <span class="icon is-
small is-right"> <i class="fas fa-exclamation-triangle"></i> </span>
@enderror </div> @error('name') <p class="help is-danger">{{ $message }}
</p> @enderror </div>
```

Further reading

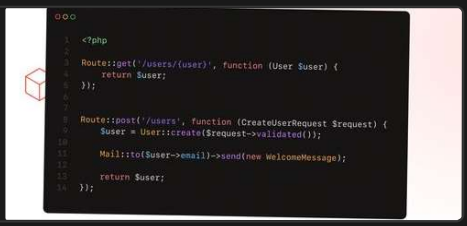
All the available validation rules

The following link leads you to a section of the Laravel documentation where you can find all the available validation rules:

Laravel - The PHP Framework For Web Artisans

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

https://laravel.com/docs/validation#available-validation-rul...



Flashing other feedback

For usability it is good practice to give feedback to your users, even when an action is successful. We've all seen these messages in other websites like: "Item added to you cart" or "Message is sent".

To implement this, a web application must store this message somewhere before redirecting so it can be rendered when the user is redirected. This technique is called *session flashing* because the session is used to store this data. The following chapter in the Laravel documentation shows you how it's done:

HTTP Responses

Strings & Arrays All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string

https://laravel.com/docs/responses#redirecting-with-flashed-session-data

If you use Bulma, you can use the *notification* element for this. See:

Notification

Bulma is a free, open source CSS framework based on Flexbox and built with Sass. It's 100% responsive, fully modular, and

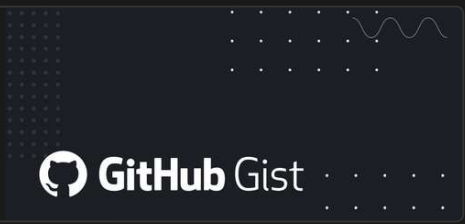
https://bulma.io/documentation/elements/notification/

The following gist shows my standard notifications code when using Bulma:

Bulma components for standard Laravel messages

Instantly share code, notes, and snippets. Bulma components for standard Laravel messages You can't perform that action at this

https://gist.github.com/dwaard/2051000c03c0130b94ad63...



You can use an `@include()` directive in your views to include this easily.

JavaScript form validation

Remember, **server-side validation** is an absolute necessity, primarily for security purposes, and while it complements usability to some extent, it stands as the sole indispensable form of validation in this course. You are allowed to use client side validation, but this must always be in combination with server side

Check the following YouTube video if you want to know more about client side form validation with JavaScript: