Forms

Because forms are the backbone of most web applications, Livewire provides loads of helpful utilities for building them. From handling simple input elements to complex things like real-time validation or file uploading, Livewire has simple, well-documented tools to make your life easier and delight your users.

Let's dive in.

Submitting a form

Let's start by looking at a very simple form in a `CreatePost` component. This form will have two simple text inputs and a submit button, as well as some code on the backend to manage the form's state and submission:

```php
<?php

namespace App\Livewire;

use Livewire\Component;
use App\Models\Post;

class CreatePost extends Component
```

```
{
    public $title = '';

    public $content = '';

    public function save()
    {
        Post::create(
            $this->only(['title', 'content'])
        );

        session()->flash('status', 'Post successfully updated.');

        return $this->redirect('/posts');
    }

    public function render()
    {
        return view('livewire.create-post');
    }
}
```

```
<form wire:submit="save">
    <input type="text" wire:model="title">

    <input type="text" wire:model="content">

    <button type="submit">Save</button>
</form>
```

As you can see, we are "binding" the public **`$title`** and **`$content`** properties in the form above using **`wire:model`**. This is one of the most commonly used and powerful features of Livewire.

In addition to binding **`$title`** and **`$content`**, we are using **`wire:submit`** to capture the **`submit`** event when the "Save" button is clicked and invoking the **`save()`** action. This action will persist the form input to the database.

After the new post is created in the database, we redirect the user to the **`ShowPosts`** component page and show them a "flash" message that the new post was created.

Adding validation

To avoid storing incomplete or dangerous user input, most forms need some sort of input validation.

Livewire makes validating your forms as simple as adding **`#[Validate]`** attributes above the properties you want to be validated.

Once a property has a **`#[Validate]`** attribute attached to it, the validation rule will be applied to the property's value any time it's updated server-side.

Let's add some basic validation rules to the **`$title`** and **`$content`** properties in our **`CreatePost`** component:

```php
<?php

namespace App\Livewire;

use Livewire\Attributes\Validate;
use Livewire\Component;
use App\Models\Post;

class CreatePost extends Component
{
    #[Validate('required')]
    public $title = '';

    #[Validate('required')]
    public $content = '';

    public function save()
    {
        $this->validate();
```

```
            Post::create(
                $this->only(['title', 'content'])
            );

            return $this->redirect('/posts');
        }

        public function render()
        {
            return view('livewire.create-post');
        }
    }
```

We'll also modify our Blade template to show any validation errors on the page.

```
    <form wire:submit="save">
        <input type="text" wire:model="title">
        <div>
            @error('title') <span class="error">{{ $message }}</span> @enderr
        </div>

        <input type="text" wire:model="content">
        <div>
            @error('content') <span class="error">{{ $message }}</span> @ende
        </div>

        <button type="submit">Save</button>
    </form>
```

Now, if the user tries to submit the form without filling in any of the fields, they will see validation messages telling them which fields are required before saving the post.

Livewire has a lot more validation features to offer. For more information, visit our **dedicated documentation page on Validation**.

Extracting a form object

If you are working with a large form and prefer to extract all of its properties, validation logic, etc., into a separate class, Livewire offers form objects.

Form objects allow you to re-use form logic across components and provide a nice way to keep your component class cleaner by grouping all form-related code into a separate class.

You can either create a form class by hand or use the convenient artisan command:

```
php artisan livewire:form PostForm
```

The above command will create a file called **`app/Livewire/Forms/PostForm.php`**.

Let's rewrite the **`CreatePost`** component to use a **`PostForm`** class:

```
<?php

namespace App\Livewire\Forms;

use Livewire\Attributes\Validate;
use Livewire\Form;

class PostForm extends Form
{
    #[Validate('required|min:5')]
    public $title = '';

    #[Validate('required|min:5')]
    public $content = '';
}
```

```
<?php

namespace App\Livewire;

use App\Livewire\Forms\PostForm;
```

```php
use Livewire\Component;
use App\Models\Post;

class CreatePost extends Component
{
    public PostForm $form;

    public function save()
    {
        $this->validate();

        Post::create(
            $this->form->all()
        );

        return $this->redirect('/posts');
    }

    public function render()
    {
        return view('livewire.create-post');
    }
}
```

```blade
<form wire:submit="save">
    <input type="text" wire:model="form.title">
    <div>
        @error('form.title') <span class="error">{{ $message }}</span> @e
    </div>

    <input type="text" wire:model="form.content">
    <div>
        @error('form.content') <span class="error">{{ $message }}</span>
    </div>

    <button type="submit">Save</button>
</form>
```

If you'd like, you can also extract the post creation logic into the form object like so:

```php
<?php

namespace App\Livewire\Forms;

use Livewire\Attributes\Validate;
use App\Models\Post;
use Livewire\Form;

class PostForm extends Form
{
    #[Validate('required|min:5')]
    public $title = '';

    #[Validate('required|min:5')]
    public $content = '';

    public function store()
    {
        $this->validate();

        Post::create($this->all());
    }
}
```

Now you can call **`$this->form->store()`** from the component:

```php
class CreatePost extends Component
{
    public PostForm $form;

    public function save()
    {
        $this->form->store();

        return $this->redirect('/posts');
    }
```

```
    // ...
}
```

If you want to use this form object for both a create and update form, you can easily adapt it to handle both use cases.

Here's what it would look like to use this same form object for an **`UpdatePost`** component and fill it with initial data:

```php
<?php

namespace App\Livewire;

use App\Livewire\Forms\PostForm;
use Livewire\Component;
use App\Models\Post;

class UpdatePost extends Component
{
    public PostForm $form;

    public function mount(Post $post)
    {
        $this->form->setPost($post);
    }

    public function save()
    {
        $this->form->update();

        return $this->redirect('/posts');
    }

    public function render()
    {
        return view('livewire.create-post');
    }
```

```
}
```

```
<?php

namespace App\Livewire\Forms;

use Livewire\Attributes\Validate;
use Livewire\Form;
use App\Models\Post;

class PostForm extends Form
{
    public ?Post $post;

    #[Validate('required|min:5')]
    public $title = '';

    #[Validate('required|min:5')]
    public $content = '';

    public function setPost(Post $post)
    {
        $this->post = $post;

        $this->title = $post->title;

        $this->content = $post->content;
    }

    public function store()
    {
        $this->validate();

        Post::create($this->only(['title', 'content']));
    }

    public function update()
    {
        $this->validate();
```

```
            $this->post->update(
                $this->all()
            );
        }
    }
```

As you can see, we've added a `setPost()` method to the `PostForm` object to optionally allow for filling the form with existing data as well as storing the post on the form object for later use. We've also added an `update()` method for updating the existing post.

Form objects are not required when working with Livewire, but they do offer a nice abstraction for keeping your components free of repetitive boilerplate.

Resetting form fields

If you are using a form object, you may want to reset the form after it has been submitted. This can be done by calling the `reset()` method:

```
<?php

namespace App\Livewire\Forms;

use Livewire\Attributes\Validate;
use App\Models\Post;
use Livewire\Form;

class PostForm extends Form
{
    #[Validate('required|min:5')]
    public $title = '';

    #[Validate('required|min:5')]
    public $content = '';

    // ...
```

```
    public function store()
    {
        $this->validate();

        Post::create($this->all());

        $this->reset();
    }
}
```

You can also reset specific properties by passing the property names into the **`reset()`** method:

```
$this->reset('title');

// Or multiple at once...

$this->reset(['title', 'content']);
```

Pulling form fields

Alternatively, you can use the **`pull()`** method to both retrieve a form's properties and reset them in one operation.

```
<?php

namespace App\Livewire\Forms;

use Livewire\Attributes\Validate;
use App\Models\Post;
use Livewire\Form;

class PostForm extends Form
{
    #[Validate('required|min:5')]
    public $title = '';
```

```
    #[Validate('required|min:5')]
    public $content = '';

    // ...

    public function store()
    {
        $this->validate();

        Post::create(
            $this->pull()
        );
    }
}
```

You can also pull specific properties by passing the property names into the **`pull()`** method:

```
// Return a value before resetting...
$this->pull('title');

 // Return a key-value array of properties before resetting...
$this->pull(['title', 'content']);
```

Using Rule objects

If you have more sophisticated validation scenarios where Laravel's **`Rule`** objects are necessary, you can alternatively define a **`rules()`** method to declare your validation rules like so:

```
<?php

namespace App\Livewire\Forms;

use Illuminate\Validation\Rule;
use App\Models\Post;
```