

filament

Filament 3.x

Offline Documentation

Last updated on: Mon Jul 29 2024

Created & maintained by — [Mohammad Nurul Islam \(Shihan\)](#).

Preface

Why PDF version?

As a full stack web (and backend service) developer, I often need to create api & admin interface for small to medium web applications in the shortest possible time. And due to expressive syntax & the rapid development tools provided, I find [Laravel](#) to be very useful for this purpose.

Also, for myself I sometimes intentionally turn off internet while writing code to focus on work to meet deadline. During that isolated time, I need offline versions of documentation of various tools, libraries and framework. Even though, for most of the other tools, libraries & frameworks offline documentation is available, for Laravel it was never true. To solve this issue, few years back I created & maintained offline downloadable documentation of all Laravel versions available then and published in [laravel-docs-in-pdf](#) through scripted processing on the markdown files from the [Laravel documentation repo](#) available in github.

Nowadays, it is almost impractical to start working in a Laravel project without installing [Filament](#) packages. And again, I feel the same issue of no offline documentation for [Filament](#). ☺

As a result, I've decided to create offline (PDF) documentation for [Filament](#) and here is the result.

In [filament-docs-in-pdf](#) repo, you can always find up-to-date PDF files corresponding to documentation for each version of [Filament](#) and download copies for offline use. If you find these PDF documentation files useful and want to show some support, please give [the repo](#) a star and share with your social/professional network.

You can also [watch the repo](#) and/or [follow me](#) on [Github](#), [Twitter](#) or [Linkedin](#), to get [notification](#) about future releases of the PDF version of Laravel documentation.

Any kind of **appreciation** from the community will **encourage & motivate me** to continue maintaining this project **regularly**. To show your generous support, you can of course



Buy me a coffee

License & copyright

All the PDF files available in [filament-docs-in-pdf](#) repo are licensed under [The Unlicense](#) license and available in [public domain](#). Feel free to copy, modify, distribute or whatever you want to do with these files.

But, please be aware that, the documentation content of all the PDF files (beside the cover & preface pages) are not covered with this license and is licensed under the original license of [Filament documentation](#) as defined in the [original documentation's repo](#).

Table of Contents

- **Panel Builder**

- [Installation](#)
- [Getting Started](#)
- **Resources**
 - [Getting Started](#)
 - [Listing Records](#)
 - [Creating Records](#)
 - [Editing Records](#)
 - [Viewing Records](#)
 - [Deleting Records](#)
 - [Relation Managers](#)
 - [Global Search](#)
 - [Widgets](#)
 - [Custom Pages](#)
 - [Security](#)
- [Pages](#)
- [Dashboard](#)
- [Navigation](#)
- [Notifications](#)
- [Users](#)
- [Configuration](#)
- [Clusters](#)
- [Tenancy](#)
- [Themes](#)
- [Plugins](#)
- [Testing](#)
- [Upgrade Guide](#)

- **Form Builder**

- [Installation](#)
- [Getting Started](#)
- **Fields**
 - [Getting Started](#)
 - [Text Input](#)
 - [Select](#)
 - [Checkbox](#)
 - [Toggle](#)
 - [Checkbox List](#)
 - [Radio](#)
 - [Date Time Picker](#)
 - [File Upload](#)
 - [Rich Editor](#)
 - [Markdown Editor](#)
 - [Repeater](#)
 - [Builder](#)
 - [Tags Input](#)
 - [Textarea](#)

- [Key Value](#)
- [Color Picker](#)
- [Toggle Buttons](#)
- [Hidden](#)
- [Custom](#)

- **[Layout](#)**

- [Getting Started](#)
- [Grid](#)
- [Fieldset](#)
- [Tabs](#)
- [Wizard](#)
- [Section](#)
- [Split](#)
- [Custom](#)
- [Placeholder](#)
- [Validation](#)
- [Actions](#)
- [Advanced](#)
- [Adding A Form To A Livewire Component](#)
- [Testing](#)
- [Upgrade Guide](#)

- **[Table Builder](#)**

- [Installation](#)
- [Getting Started](#)
- **[Columns](#)**
 - [Getting Started](#)
 - [Text](#)
 - [Icon](#)
 - [Image](#)
 - [Color](#)
 - [Select](#)
 - [Toggle](#)
 - [Text Input](#)
 - [Checkbox](#)
 - [Custom](#)
 - [Relationships](#)
 - [Advanced](#)
- **[Filters](#)**
 - [Getting Started](#)
 - [Select](#)
 - [Ternary](#)
 - [Query Builder](#)
 - [Custom](#)
 - [Layout](#)
- [Actions](#)
- [Layout](#)
- [Summaries](#)
- [Grouping](#)
- [Empty State](#)
- [Advanced](#)

- [Adding A Table To A Livewire Component](#)
- [Testing](#)
- [Upgrade Guide](#)

• **Notifications**

- [Installation](#)
- [Sending Notifications](#)
- [Database Notifications](#)
- [Broadcast Notifications](#)
- [Customizing Notifications](#)
- [Testing](#)
- [Upgrade Guide](#)

• **Actions**

- [Installation](#)
- [Overview](#)
- [Trigger Button](#)
- [Modals](#)
- [Grouping Actions](#)
- [Adding An Action To A Livewire Component](#)
- **Prebuilt Actions**
 - [Create](#)
 - [Edit](#)
 - [View](#)
 - [Delete](#)
 - [Replicate](#)
 - [Force Delete](#)
 - [Restore](#)
 - [Import](#)
 - [Export](#)
- [Advanced](#)
- [Testing](#)
- [Upgrade Guide](#)

• **Infolist Builder**

- [Installation](#)
- [Getting Started](#)
- **Entries**
 - [Getting Started](#)
 - [Text](#)
 - [Icon](#)
 - [Image](#)
 - [Color](#)
 - [Key Value](#)
 - [Repeatable](#)
 - [Custom](#)
- **Layout**
 - [Getting Started](#)
 - [Grid](#)
 - [Fieldset](#)

- [Tabs](#)
- [Section](#)
- [Split](#)
- [Custom](#)
- [Actions](#)
- [Advanced](#)
- [Adding An Infolist To A Livewire Component](#)
- [Testing](#)

- **Widgets**

- [Installation](#)
- [Stats Overview](#)
- [Charts](#)
- [Tables](#)
- [Adding A Widget To A Blade View](#)

- **Core Concepts**

- [Overview](#)
- [Assets](#)
- [Icons](#)
- [Colors](#)
- [Style Customization](#)
- [Render Hooks](#)
- [Enums](#)
- [Contributing](#)
- **Plugins**
 - [Getting Started](#)
 - [Build A Panel Plugin](#)
 - [Build A Standalone Plugin](#)

- **Blade Components**

- [Overview](#)
- [Avatar](#)
- [Badge](#)
- [Breadcrumbs](#)
- [Button](#)
- [Checkbox](#)
- [Dropdown](#)
- [Fieldset](#)
- [Icon Button](#)
- [Input Wrapper](#)
- [Input](#)
- [Link](#)
- [Loading Indicator](#)
- [Modal](#)
- [Pagination](#)
- [Section](#)
- [Select](#)
- [Tabs](#)

- [Stubs](#)
- [Support](#)

Chapter 1

Panel Builder

Installation

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

If you are upgrading from Filament v2, please review the [upgrade guide](#).

Install the Filament Panel Builder by running the following commands in your Laravel project directory:

```
composer require filament/filament:"^3.2" -W
```

```
php artisan filament:install --panels
```

This will create and register a new [Laravel service provider](#) called

```
app/Providers/Filament/AdminPanelProvider.php
```

If you get an error when accessing your panel, check that the service provider was registered in [bootstrap/providers.php](#) (Laravel 11 and above) or [config/app.php](#) (Laravel 10 and below). If not, you should manually add it.

Create a user

You can create a new user account with the following command:

```
php artisan make:filament-user
```

Open [/admin](#) in your web browser, sign in, and start building your app!

Not sure where to start? Review the [Getting Started guide](#) to learn how to build a complete Filament admin panel.

Using other Filament packages

The Filament Panel Builder pre-installs the [Form Builder](#), [Table Builder](#), [Notifications](#), [Actions](#), [Infolists](#), and [Widgets](#) packages. No other installation steps are required to use these packages within a panel.

Improving Filament panel performance

Optimizing Filament for production

To optimize Filament for production, you should run the following command in your deployment script:

```
php artisan filament:optimize
```

This command will [cache the Filament components](#) and additionally the [Blade icons](#), which can significantly improve the performance of your Filament panels. This command is a shorthand for the commands [php artisan filament:cache-](#)

`components` and `php artisan icons:cache`.

To clear the caches at once, you can run:

```
php artisan filament:optimize-clear
```

Caching Filament components

If you're not using the `filament:optimize` command, you may wish to consider running `php artisan filament:cache-components` in your deployment script, especially if you have large numbers of components (resources, pages, widgets, relation managers, custom Livewire components, etc.). This will create cache files in the `bootstrap/cache/filament` directory of your application, which contain indexes for each type of component. This can significantly improve the performance of Filament in some apps, as it reduces the number of files that need to be scanned and auto-discovered for components.

However, if you are actively developing your app locally, you should avoid using this command, as it will prevent any new components from being discovered until the cache is cleared or rebuilt.

You can clear the cache at any time without rebuilding it by running `php artisan filament:clear-cached-components`.

Caching Blade Icons

If you're not using the `filament:optimize` command, you may wish to consider running `php artisan icons:cache` locally, and also in your deployment script. This is because Filament uses the [Blade Icons](#) package, which can be much more performant when cached.

Enabling OPcache on your server

From the [Laravel Forge documentation](#):

Optimizing the PHP OPcache for production will configure OPcache to store your compiled PHP code in memory to greatly improve performance.

Please use a search engine to find the relevant OPcache setup instructions for your environment.

Optimizing your Laravel app

You should also consider optimizing your Laravel app for production by running `php artisan optimize` in your deployment script. This will cache the configuration files and routes.

Deploying to production

Allowing users to access a panel

By default, all `User` models can access Filament locally. However, when deploying to production, you must update your `App\Models\User.php` to implement the `FilamentUser` contract — ensuring that only the correct users can access your panel:

```

<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Panel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements FilamentUser
{
    // ...

    public function canAccessPanel(Panel $panel): bool
    {
        return str_ends_with($this->email, '@yourdomain.com') && $this->hasVerifiedEmail();
    }
}

```

If you don't complete these steps, a 403 Forbidden error will be returned when accessing the app in production.

Learn more about [users](#).

Publishing configuration

You can publish the Filament package configuration (if needed) using the following command:

```
php artisan vendor:publish --tag=filament-config
```

Publishing translations

You can publish the language files for translations (if needed) with the following command:

```
php artisan vendor:publish --tag=filament-panels-translations
```

Since this package depends on other Filament packages, you can publish the language files for those packages with the following commands:

```

php artisan vendor:publish --tag=filament-actions-translations

php artisan vendor:publish --tag=filament-forms-translations

php artisan vendor:publish --tag=filament-infolists-translations

php artisan vendor:publish --tag=filament-notifications-translations

php artisan vendor:publish --tag=filament-tables-translations

php artisan vendor:publish --tag=filament-translations

```

Upgrading

If you're upgrading from Filament v2? Please review the [upgrade guide](#).

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [  
    // ...  
    "@php artisan filament:upgrade"  
,
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post-autoload-dump` hook, you can run the command yourself:

```
composer update  
  
php artisan filament:upgrade
```

Getting Started

Overview

Panels are the top-level container in Filament, allowing you to build feature-rich admin panels that include pages, resources, forms, tables, notifications, actions, infolists, and widgets. All Panels include a default dashboard that can include widgets with statistics, charts, tables, and more.

Prerequisites

Before using Filament, you should be familiar with Laravel. Filament builds upon many core Laravel concepts, especially [database migrations](#) and [Eloquent ORM](#). If you're new to Laravel or need a refresher, we highly recommend completing the [Laravel Bootcamp](#), which covers the fundamentals of building Laravel apps.

The demo project

This guide covers building a simple patient management system for a veterinary practice using Filament. It will support adding new patients (cats, dogs, or rabbits), assigning them to an owner, and recording which treatments they received. The system will have a dashboard with statistics about the types of patients and a chart showing the number of treatments administered over the past year.

Setting up the database and models

This project needs three models and migrations: `Owner`, `Patient`, and `Treatment`. Use the following artisan commands to create these:

```
php artisan make:model Owner -m
php artisan make:model Patient -m
php artisan make:model Treatment -m
```

Defining migrations

Use the following basic schemas for your database migrations:

```
// create_owners_table
Schema::create('owners', function (Blueprint $table) {
    $table->id();
    $table->string('email');
    $table->string('name');
    $table->string('phone');
    $table->timestamps();
});

// create_patients_table
Schema::create('patients', function (Blueprint $table) {
    $table->id();
    $table->date('date_of_birth');
    $table->string('name');
    $table->foreignId('owner_id')->constrained('owners')->cascadeOnDelete();
    $table->string('type');
    $table->timestamps();
});

// create_treatments_table
Schema::create('treatments', function (Blueprint $table) {
    $table->id();
    $table->string('description');
    $table->text('notes')->nullable();
    $table->foreignId('patient_id')->constrained('patients')->cascadeOnDelete();
    $table->unsignedInteger('price')->nullable();
    $table->timestamps();
});
```

Run the migrations using `php artisan migrate`.

Unguarding all models

For brevity in this guide, we will disable Laravel's [mass assignment protection](#). Filament only saves valid data to models so the models can be unguarded safely. To unguard all Laravel models at once, add `Model::unguard()` to the `boot()` method of `app/Providers/AppServiceProvider.php`:

```
use Illuminate\Database\Eloquent\Model;

public function boot(): void
{
    Model::unguard();
}
```

Setting up relationships between models

Let's set up relationships between the models. For our system, pet owners can own multiple pets (patients), and patients can have many treatments:

```

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Owner extends Model
{
    public function patients(): HasMany
    {
        return $this->hasMany(Patient::class);
    }
}

class Patient extends Model
{
    public function owner(): BelongsTo
    {
        return $this->belongsTo(Owner::class);
    }

    public function treatments(): HasMany
    {
        return $this->hasMany(Treatment::class);
    }
}

class Treatment extends Model
{
    public function patient(): BelongsTo
    {
        return $this->belongsTo(Patient::class);
    }
}

```

Introducing resources

In Filament, resources are static classes used to build CRUD interfaces for your Eloquent models. They describe how administrators can interact with data from your panel using tables and forms.

Since patients (pets) are the core entity in this system, let's start by creating a patient resource that enables us to build pages for creating, viewing, updating, and deleting patients.

Use the following artisan command to create a new Filament resource for the `Patient` model:

```
php artisan make:filament-resource Patient
```

This will create several files in the `app/Filament/Resources` directory:

```
.
+-- PatientResource.php
+-- PatientResource
|   +-- Pages
|   |   +-- CreatePatient.php
|   |   +-- EditPatient.php
|   |   +-- ListPatients.php
```

Visit `/admin/patients` in your browser and observe a new link called "Patients" in the navigation. Clicking the link will display an empty table. Let's add a form to create new patients.

Setting up the resource form

If you open the `PatientResource.php` file, there's a `form()` method with an empty `schema([...])` array. Adding form fields to this schema will build a form that can be used to create and edit new patients.

"Name" text input

Filament bundles a large selection of form fields. Let's start with a simple text input field:

```
use Filament\Forms;
use Filament\Forms\Form;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name'),
        ]);
}
```

Visit `/admin/patients/create` (or click the "New Patient" button) and observe that a form field for the patient's name was added.

Since this field is required in the database and has a maximum length of 255 characters, let's add two validation rules to the name field:

```
use Filament\Forms;

Forms\Components\TextInput::make('name')
    ->required()
    ->maxLength(255)
```

Attempt to submit the form to create a new patient without a name and observe that a message is displayed informing you that the name field is required.

"Type" select

Let's add a second field for the type of patient: a choice between a cat, dog, or rabbit. Since there's a fixed set of options to choose from, a select field works well:

```

use Filament\Forms;
use Filament\Forms\Form;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')
                ->required()
                ->maxLength(255),
            Forms\Components\Select::make('type')
                ->options([
                    'cat' => 'Cat',
                    'dog' => 'Dog',
                    'rabbit' => 'Rabbit',
                ]),
        ]);
}

```

The `options()` method of the Select field accepts an array of options for the user to choose from. The array keys should match the database, and the values are used as the form labels. Feel free to add as many animals to this array as you wish.

Since this field is also required in the database, let's add the `required()` validation rule:

```

use Filament\Forms;

Forms\Components\Select::make('type')
    ->options([
        'cat' => 'Cat',
        'dog' => 'Dog',
        'rabbit' => 'Rabbit',
    ])
    ->required()

```

"Date of birth" picker

Let's add a [date picker field](#) for the `date_of_birth` column along with the validation (the date of birth is required and the date should be no later than the current day).

```

use Filament\Forms;
use Filament\Forms\Form;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')
                ->required()
                ->maxLength(255),
            Forms\Components\Select::make('type')
                ->options([
                    'cat' => 'Cat',
                    'dog' => 'Dog',
                    'rabbit' => 'Rabbit',
                ])
                ->required(),
            Forms\Components\DatePicker::make('date_of_birth')
                ->required()
                ->maxDate(now()),
        ]);
}

```

"Owner" select

We should also add an owner when creating a new patient. Since we added a `BelongsTo` relationship in the Patient model (associating it to the related `Owner` model), we can use the `relationship()` method from the select field to load a list of owners to choose from:

```

use Filament\Forms;
use Filament\Forms\Form;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')
                ->required()
                ->maxLength(255),
            Forms\Components\Select::make('type')
                ->options([
                    'cat' => 'Cat',
                    'dog' => 'Dog',
                    'rabbit' => 'Rabbit',
                ])
                ->required(),
            Forms\Components\DatePicker::make('date_of_birth')
                ->required()
                ->maxDate(now()),
            Forms\Components\Select::make('owner_id')
                ->relationship('owner', 'name')
                ->required(),
        ]);
}

```

The first argument of the `relationship()` method is the name of the function that defines the relationship in the model (used by Filament to load the select options) — in this case, `owner`. The second argument is the column name to use from the related table — in this case, `name`.

Let's also make the `owner` field required, `searchable()`, and `preload()` the first 50 owners into the searchable list (in case the list is long):

```
use Filament\Forms;

Forms\Components\Select::make('owner_id')
    ->relationship('owner', 'name')
    ->searchable()
    ->preload()
    ->required()
```

Creating new owners without leaving the page

Currently, there are no owners in our database. Instead of creating a separate Filament owner resource, let's give users an easier way to add owners via a modal form (accessible as a `[+]` button next to the select). Use the `createOptionForm()` method to embed a modal form with `TextInput` fields for the owner's name, email address, and phone number:

```
use Filament\Forms;

Forms\Components\Select::make('owner_id')
    ->relationship('owner', 'name')
    ->searchable()
    ->preload()
    ->createOptionForm([
        Forms\Components\TextInput::make('name')
            ->required()
            ->maxLength(255),
        Forms\Components\TextInput::make('email')
            ->label('Email address')
            ->email()
            ->required()
            ->maxLength(255),
        Forms\Components\TextInput::make('phone')
            ->label('Phone number')
            ->tel()
            ->required(),
    ])
    ->required()
```

A few new methods on the `TextInput` were used in this example:

- `label()` overrides the auto-generated label for each field. In this case, we want the `Email` label to be `Email address`, and the `Phone` label to be `Phone number`.
- `email()` ensures that only valid email addresses can be input into the field. It also changes the keyboard layout on mobile devices.
- `tel()` ensures that only valid phone numbers can be input into the field. It also changes the keyboard layout on mobile devices.

The form should be working now! Try creating a new patient and their owner. Once created, you will be redirected to the Edit page, where you can update their details.

Setting up the patients table

Visit the `/admin/patients` page again. If you have created a patient, there should be one empty row in the table — with an edit button. Let's add some columns to the table, so we can view the actual patient data.

Open the `PatientResource.php` file. You should see a `table()` method with an empty `columns([...])` array. You can use this array to add columns to the `patients` table.

Adding text columns

Filament bundles a large selection of table columns. Let's use a simple text column for all the fields in the `patients` table:

```
use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('name'),
            Tables\Columns\TextColumn::make('type'),
            Tables\Columns\TextColumn::make('date_of_birth'),
            Tables\Columns\TextColumn::make('owner.name'),
        ]);
}
```

Filament uses dot notation to eager-load related data. We used `owner.name` in our table to display a list of owner names instead of less informational ID numbers. You could also add columns for the owner's email address and phone number.

Making columns searchable

The ability to search for patients directly in the table would be helpful as a veterinary practice grows. You can make columns searchable by chaining the `searchable()` method to the column. Let's make the patient's name and owner's name searchable.

```
use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('name')
                ->searchable(),
            Tables\Columns\TextColumn::make('type'),
            Tables\Columns\TextColumn::make('date_of_birth'),
            Tables\Columns\TextColumn::make('owner.name')
                ->searchable(),
        ]);
}
```

Reload the page and observe a new search input field on the table that filters the table entries using the search criteria.

Making the columns sortable

To make the `patients` table sortable by age, add the `sortable()` method to the `date_of_birth` column:

```

use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('name')
                ->searchable(),
            Tables\Columns\TextColumn::make('type'),
            Tables\Columns\TextColumn::make('date_of_birth')
                ->sortable(),
            Tables\Columns\TextColumn::make('owner.name')
                ->searchable(),
        ]);
}

```

This will add a sort icon button to the column header. Clicking it will sort the table by date of birth.

Filtering the table by patient type

Although you can make the `type` field searchable, making it filterable is a much better user experience.

Filament tables can have [filters](#), which are components that reduce the number of records in a table by adding a scope to the Eloquent query. Filters can even contain custom form components, making them a potent tool for building interfaces.

Filament includes a prebuilt `SelectFilter` that you can add to the table's `filters()`:

```

use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters([
            Tables\Filters\SelectFilter::make('type')
                ->options([
                    'cat' => 'Cat',
                    'dog' => 'Dog',
                    'rabbit' => 'Rabbit',
                ]),
        ]);
}

```

Reload the page, and you should see a new filter icon in the top right corner (next to the search form). The filter opens a select menu with a list of patient types. Try filtering your patients by type.

Introducing relation managers

Currently, patients can be associated with their owners in our system. But what happens if we want a third level? Patients come to the vet practice for treatment, and the system should be able to record these treatments and associate them with a patient.

One option is to create a new `TreatmentResource` with a select field to associate treatments with a patient. However, managing treatments separately from the rest of the patient information is cumbersome for the user. Filament uses "relation managers" to solve this problem.

Relation managers are tables that display related records for an existing resource on the edit screen for the parent resource. For example, in our project, you could view and manage a patient's treatments directly below their edit form.

You can also use Filament `"actions"` to open a modal form to create, edit, and delete treatments directly from the patient's table.

Use the `make:filament-relation-manager` artisan command to quickly create a relation manager, connecting the patient resource to the related treatments:

```
php artisan make:filament-relation-manager PatientResource treatments description
```

- `PatientResource` is the name of the resource class for the owner model. Since treatments belong to patients, the treatments should be displayed on the Edit Patient page.
- `treatments` is the name of the relationship in the Patient model we created earlier.
- `description` is the column to display from the treatments table.

This will create a `PatientResource/RelationManagers/TreatmentsRelationManager.php` file. You must register the new relation manager in the `getRelations()` method of the `PatientResource`:

```
use App\Filament\Resources\PatientResource\RelationManagers;

public static function getRelations(): array
{
    return [
        RelationManagers\TreatmentsRelationManager::class,
    ];
}
```

The `TreatmentsRelationManager.php` file contains a class that is prepopulated with a form and table using the parameters from the `make:filament-relation-manager` artisan command. You can customize the fields and columns in the relation manager similar to how you would in a resource:

```

use Filament\Forms;
use Filament\Forms\Form;
use Filament\Tables;
use Filament\Tables\Table;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('description')
                ->required()
                ->maxLength(255),
        ]);
}

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('description'),
        ]);
}

```

Visit the Edit page for one of your patients. You should already be able to create, edit, delete, and list treatments for that patient.

Setting up the treatment form

By default, text fields only span half the width of the form. Since the `description` field might contain a lot of information, add a `columnSpan('full')` method to make the field span the entire width of the modal form:

```

use Filament\Forms;

Forms\Components\TextInput::make('description')
    ->required()
    ->maxLength(255)
    ->columnSpan('full')

```

Let's add the `notes` field, which can be used to add more details about the treatment. We can use a `textarea` field with a `columnSpan('full')`:

```
use Filament\Forms;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('description')
                ->required()
                ->maxLength(255)
                ->columnSpan('full'),
            Forms\Components\Textarea::make('notes')
                ->maxLength(65535)
                ->columnSpan('full'),
        ]);
}
```

Configuring the `price` field

Let's add a `price` field for the treatment. We can use a text input with some customizations to make it suitable for currency input. It should be `numeric()`, which adds validation and changes the keyboard layout on mobile devices. Add your preferred currency prefix using the `prefix()` method; for example, `prefix('€')` will add a `€` before the input without impacting the saved output value:

```
use Filament\Forms;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('description')
                ->required()
                ->maxLength(255)
                ->columnSpan('full'),
            Forms\Components\Textarea::make('notes')
                ->maxLength(65535)
                ->columnSpan('full'),
            Forms\Components\TextInput::make('price')
                ->numeric()
                ->prefix('€')
                ->maxValue(42949672.95),
        ]);
}
```

Casting the price to an integer

Filament stores currency values as integers (not floats) to avoid rounding and precision issues — a widely-accepted approach in the Laravel community. However, this requires creating a cast in Laravel that transforms the integer into a float when retrieved and back to an integer when stored in the database. Use the following artisan command to create the cast:

```
php artisan make:cast MoneyCast
```

Inside the new `app/Casts/MoneyCast.php` file, update the `get()` and `set()` methods:

```

public function get($model, string $key, $value, array $attributes): float
{
    // Transform the integer stored in the database into a float.
    return round(floatval($value) / 100, precision: 2);
}

public function set($model, string $key, $value, array $attributes): float
{
    // Transform the float into an integer for storage.
    return round(floatval($value) * 100);
}

```

Now, add the `MoneyCast` to the `price` attribute in the `Treatment` model:

```

use App\Cast\Cast;
use Illuminate\Database\Eloquent\Model;

class Treatment extends Model
{
    protected $casts = [
        'price' => MoneyCast::class,
    ];

    // ...
}

```

Setting up the treatments table

When the relation manager was generated previously, the `description` text column was automatically added. Let's also add a `sortable()` column for the `price` with a currency prefix. Use the Filament `money()` method to format the `price` column as money — in this case for `EUR` (`€`):

```

use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('description'),
            Tables\Columns\TextColumn::make('price')
                ->money('EUR')
                ->sortable(),
        ]);
}

```

Let's also add a column to indicate when the treatment was administered using the default `created_at` timestamp. Use the `dateTime()` method to display the date-time in a human-readable format:

```

use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('description'),
            Tables\Columns\TextColumn::make('price')
                ->money('EUR')
                ->sortable(),
            Tables\Columns\TextColumn::make('created_at')
                ->dateTime(),
        ]);
}

```

You can pass any valid [PHP date formatting string](#) to the `dateTime()` method (e.g. `dateTime('m-d-Y h:i A')`).

Introducing widgets

Filament widgets are components that display information on your dashboard, especially statistics. Widgets are typically added to the default [Dashboard](#) of the panel, but you can add them to any page, including resource pages. Filament includes built-in widgets like the [stats widget](#), to render important statistics in a simple overview; [chart widget](#), which can render an interactive chart; and [table widget](#), which allows you to easily embed the Table Builder.

Let's add a stats widget to our default dashboard page that includes a stat for each type of patient and a chart to visualize treatments administered over time.

Creating a stats widget

Create a [stats widget](#) to render patient types using the following artisan command:

```
php artisan make:filament-widget PatientTypeOverview --stats-overview
```

When prompted, do not specify a resource, and select "admin" for the location.

This will create a new `app/Filament/Widgets/PatientTypeOverview.php` file. Open it, and return `Stat` instances from the `getStats()` method:

```
<?php

namespace App\Filament\Widgets;

use App\Models\Patient;
use Filament\Widgets\StatsOverviewWidget as BaseWidget;
use Filament\Widgets\StatsOverviewWidget\Stat;

class PatientTypeOverview extends BaseWidget
{
    protected function getStats(): array
    {
        return [
            Stat::make('Cats', Patient::query()->where('type', 'cat')->count()),
            Stat::make('Dogs', Patient::query()->where('type', 'dog')->count()),
            Stat::make('Rabbits', Patient::query()->where('type', 'rabbit')->count()),
        ];
    }
}
```

Open your dashboard, and you should see your new widget displayed. Each stat should show the total number of patients for the specified type.

Creating a chart widget

Let's add a chart to the dashboard to visualize the number of treatments administered over time. Use the following artisan command to create a new chart widget:

```
php artisan make:filament-widget TreatmentsChart --chart
```

When prompted, do not specify a resource, select "admin" for the location, and choose "line chart" as the chart type.

Open `app/Filament/Widgets/TreatmentsChart.php` and set the `$heading` of the chart to "Treatments".

The `getData()` method returns an array of datasets and labels. Each dataset is a labeled array of points to plot on the chart, and each label is a string. This structure is identical to the [Chart.js](#) library, which Filament uses to render charts.

To populate chart data from an Eloquent model, Filament recommends that you install the [flowframe/laravel-trend](#) package:

```
composer require flowframe/laravel-trend
```

Update the `getData()` to display the number of treatments per month for the past year:

```

use App\Models\Treatment;
use Flowframe\Trend\Trend;
use Flowframe\Trend\TrendValue;

protected function getData(): array
{
    $data = Trend::model(Treatment::class)
        ->between(
            start: now()->subYear(),
            end: now(),
        )
        ->perMonth()
        ->count();

    return [
        'datasets' => [
            [
                'label' => 'Treatments',
                'data' => $data->map(fn (TrendValue $value) => $value->aggregate),
            ],
            [
                'label' => 'Treatments',
                'data' => $data->map(fn (TrendValue $value) => $value->date),
            ],
        ],
    ];
}

```

Now, check out your new chart widget in the dashboard!

You can [customize your dashboard page](#) to change the grid and how many widgets are displayed.

Next steps with the Panel Builder

Congratulations! Now that you know how to build a basic Filament application, here are some suggestions for further learning:

- [Create custom pages in the panel that don't belong to resources.](#)
- [Learn more about adding action buttons to pages and resources, with modals to collect user input or for confirmation.](#)
- [Explore the available fields to collect input from your users.](#)
- [Check out the list of form layout components.](#)
- [Discover how to build complex, responsive table layouts without touching CSS.](#)
- [Add summaries to your tables](#)
- [Write automated tests for your panel using our suite of helper methods.](#)

Resources

Getting Started

Overview

Resources are static classes that are used to build CRUD interfaces for your Eloquent models. They describe how administrators should be able to interact with data from your app - using tables and forms.

Creating a resource

To create a resource for the `App\Models\Customer` model:

```
php artisan make:filament-resource Customer
```

This will create several files in the `app/Filament/Resources` directory:

```
.
+-- CustomerResource.php
+-- CustomerResource
|   +-- Pages
|   |   +-- CreateCustomer.php
|   |   +-- EditCustomer.php
|   |   +-- ListCustomers.php
```

Your new resource class lives in `CustomerResource.php`.

The classes in the `Pages` directory are used to customize the pages in the app that interact with your resource. They're all full-page [Livewire](#) components that you can customize in any way you wish.

Have you created a resource, but it's not appearing in the navigation menu? If you have a `model_policy`, make sure you return `true` from the `viewAny()` method.

Simple (modal) resources

Sometimes, your models are simple enough that you only want to manage records on one page, using modals to create, edit and delete records. To generate a simple resource with modals:

```
php artisan make:filament-resource Customer --simple
```

Your resource will have a "Manage" page, which is a List page with modals added.

Additionally, your simple resource will have no `getRelations()` method, as [relation managers](#) are only displayed on the Edit and View pages, which are not present in simple resources. Everything else is the same.

Automatically generating forms and tables

If you'd like to save time, Filament can automatically generate the [form](#) and [table](#) for you, based on your model's database columns, using `--generate`:

```
php artisan make:filament-resource Customer --generate
```

Handling soft deletes

By default, you will not be able to interact with deleted records in the app. If you'd like to add functionality to restore, force delete and filter trashed records in your resource, use the `--soft-deletes` flag when generating the resource:

```
php artisan make:filament-resource Customer --soft-deletes
```

You can find out more about soft deleting [here](#).

Generating a View page

By default, only List, Create and Edit pages are generated for your resource. If you'd also like a [View page](#), use the `--view` flag:

```
php artisan make:filament-resource Customer --view
```

Specifying a custom model namespace

By default, Filament will assume that your model exists in the `App\Models` directory. You can pass a different namespace for the model using the `--model-namespace` flag:

```
php artisan make:filament-resource Customer --model-namespace=Custom\\Path\\Models
```

In this example, the model should exist at `Custom\Path\Models\Customer`. Please note the double backslashes `\\"` in the command that are required.

Now when [generating the resource](#), Filament will be able to locate the model and read the database schema.

Generating the model, migration and factory at the same name

If you'd like to save time when scaffolding your resources, Filament can also generate the model, migration and factory for the new resource at the same time using the `--model`, `--migration` and `--factory` flags in any combination:

```
php artisan make:filament-resource Customer --model --migration --factory
```

Record titles

A `$recordTitleAttribute` may be set for your resource, which is the name of the column on your model that can be used to identify it from others.

For example, this could be a blog post's `title` or a customer's `name`:

```
protected static ?string $recordTitleAttribute = 'name';
```

This is required for features like [global search](#) to work.

You may specify the name of an [Eloquent accessor](#) if just one column is inadequate at identifying a record.

Resource forms

Resource classes contain a `form()` method that is used to build the forms on the [Create](#) and [Edit](#) pages:

```
use Filament\Forms;
use Filament\Forms\Form;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')->required(),
            Forms\Components\TextInput::make('email')->email()->required(),
            // ...
        ]);
}
```

The `schema()` method is used to define the structure of your form. It is an array of [fields](#) and [layout components](#), in the order they should appear in your form.

Check out the Forms docs for a [guide](#) on how to build forms with Filament.

Hiding components based on the current operation

The `hiddenOn()` method of form components allows you to dynamically hide fields based on the current page or action.

In this example, we hide the `password` field on the `edit` page:

```
use Livewire\Component;

Forms\Components\TextInput::make('password')
    ->password()
    ->required()
    ->hiddenOn('edit'),
```

Alternatively, we have a `visibleOn()` shortcut method for only showing a field on one page or action:

```
use Livewire\Component;

Forms\Components\TextInput::make('password')
    ->password()
    ->required()
    ->visibleOn('create'),
```

Resource tables

Resource classes contain a `table()` method that is used to build the table on the [List](#) page:

```

use Filament\Tables;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('name'),
            Tables\Columns\TextColumn::make('email'),
            // ...
        ])
        ->filters([
            Tables\Filters\Filter::make('verified')
                ->query(fn (Builder $query): Builder => $query-
>whereNotNull('email_verified_at')),
                // ...
        ])
        ->actions([
            Tables\Actions>EditAction::make(),
        ])
        ->bulkActions([
            Tables\Actions\BulkActionGroup::make([
                Tables\Actions>DeleteBulkAction::make(),
            ]),
        ]);
}

```

Check out the [tables](#) docs to find out how to add table columns, filters, actions and more.

Authorization

For authorization, Filament will observe any [model policies](#) that are registered in your app. The following methods are used:

- `viewAny()` is used to completely hide resources from the navigation menu, and prevents the user from accessing any pages.
- `create()` is used to control [creating new records](#).
- `update()` is used to control [editing a record](#).
- `view()` is used to control [viewing a record](#).
- `delete()` is used to prevent a single record from being deleted. `deleteAny()` is used to prevent records from being bulk deleted. Filament uses the `deleteAny()` method because iterating through multiple records and checking the `delete()` policy is not very performant.
- `forceDelete()` is used to prevent a single soft-deleted record from being force-deleted. `forceDeleteAny()` is used to prevent records from being bulk force-deleted. Filament uses the `forceDeleteAny()` method because iterating through multiple records and checking the `forceDelete()` policy is not very performant.
- `restore()` is used to prevent a single soft-deleted record from being restored. `restoreAny()` is used to prevent records from being bulk restored. Filament uses the `restoreAny()` method because iterating through multiple records and checking the `restore()` policy is not very performant.
- `reorder()` is used to control [reordering a record](#).

Skipping authorization

If you'd like to skip authorization for a resource, you may set the `$shouldSkipAuthorization` property to `true`:

```
protected static bool $shouldSkipAuthorization = true;
```

Customizing the model label

Each resource has a "model label" which is automatically generated from the model name. For example, an `App\Models\Customer` model will have a `customer` label.

The label is used in several parts of the UI, and you may customize it using the `$modelLabel` property:

```
protected static ?string $modelLabel = 'cliente';
```

Alternatively, you may use the `getModelLabel()` to define a dynamic label:

```
public static function getModelLabel(): string
{
    return __('filament/resources/customer.label');
}
```

Customizing the plural model label

Resources also have a "plural model label" which is automatically generated from the model label. For example, a `customer` label will be pluralized into `customers`.

You may customize the plural version of the label using the `$pluralModelLabel` property:

```
protected static ?string $pluralModelLabel = 'clientes';
```

Alternatively, you may set a dynamic plural label in the `getPluralModelLabel()` method:

```
public static function getPluralModelLabel(): string
{
    return __('filament/resources/customer.plural_label');
}
```

Automatic model label capitalization

By default, Filament will automatically capitalize each word in the model label, for some parts of the UI. For example, in page titles, the navigation menu, and the breadcrumbs.

If you want to disable this behavior for a resource, you can set `$hasTitleCaseModelLabel` in the resource:

```
protected static bool $hasTitleCaseModelLabel = false;
```

Resource navigation items

Filament will automatically generate a navigation menu item for your resource using the [plural label](#).

If you'd like to customize the navigation item label, you may use the `$navigationLabel` property:

```
protected static ?string $navigationLabel = 'Mis Clientes';
```

Alternatively, you may set a dynamic navigation label in the `getNavigationLabel()` method:

```
public static function getNavigationLabel(): string
{
    return __('filament/resources/customer.navigation_label');
}
```

Setting a resource navigation icon

The `$navigationIcon` property supports the name of any Blade component. By default, [Heroicons](#) are installed. However, you may create your own custom icon components or install an alternative library if you wish.

```
protected static ?string $navigationIcon = 'heroicon-o-user-group';
```

Alternatively, you may set a dynamic navigation icon in the `getNavigationIcon()` method:

```
use Illuminate\Contracts\Support\Htmlable;

public static function getNavigationIcon(): string | Htmlable | null
{
    return 'heroicon-o-user-group';
}
```

Sorting resource navigation items

The `$navigationSort` property allows you to specify the order in which navigation items are listed:

```
protected static ?int $navigationSort = 2;
```

Alternatively, you may set a dynamic navigation item order in the `getNavigationSort()` method:

```
public static function getNavigationSort(): ?int
{
    return 2;
}
```

Grouping resource navigation items

You may group navigation items by specifying a `$navigationGroup` property:

```
protected static ?string $navigationGroup = 'Shop';
```

Alternatively, you may use the `getNavigationGroup()` method to set a dynamic group label:

```
public static function getNavigationGroup(): ?string
{
    return __('filament/navigation.groups.shop');
}
```

Grouping resource navigation items under other items

You may group navigation items as children of other items, by passing the label of the parent item as the `$navigationParentItem`:

```
protected static ?string $navigationParentItem = 'Products';

protected static ?string $navigationGroup = 'Shop';
```

As seen above, if the parent item has a navigation group, that navigation group must also be defined, so the correct parent item can be identified.

You may also use the `getNavigationParentItem()` method to set a dynamic parent item label:

```
public static function getNavigationParentItem(): ?string
{
    return __('filament/navigation.groups.shop.items.products');
}
```

If you're reaching for a third level of navigation like this, you should consider using [clusters](#) instead, which are a logical grouping of resources and [custom pages](#), which can share their own separate navigation.

Generating URLs to resource pages

Filament provides `getUrl()` static method on resource classes to generate URLs to resources and specific pages within them. Traditionally, you would need to construct the URL by hand or by using Laravel's `route()` helper, but these methods depend on knowledge of the resource's slug or route naming conventions.

The `getUrl()` method, without any arguments, will generate a URL to the resource's [List page](#):

```
use App\Filament\Resources\CustomerResource;

CustomerResource::getUrl(); // /admin/customers
```

You may also generate URLs to specific pages within the resource. The name of each page is the array key in the `getPages()` array of the resource. For example, to generate a URL to the [Create page](#):

```
use App\Filament\Resources\CustomerResource;

CustomerResource::getUrl('create'); // /admin/customers/create
```

Some pages in the `getPages()` method use URL parameters like `record`. To generate a URL to these pages and pass in a record, you should use the second argument:

```
use App\Filament\Resources\CustomerResource;

CustomerResource::getUrl('edit', ['record' => $customer]); // /admin/customers/edit/1
```

In this example, `$customer` can be an Eloquent model object, or an ID.

Generating URLs to resource modals

This can be especially useful if you are using [simple resources](#) with only one page.

To generate a URL for an action in the resource's table, you should pass the `tableAction` and `tableActionRecord` as URL parameters:

```
use App\Filament\Resources\CustomerResource;
use Filament\Tables\Actions>EditAction;

CustomerResource::getUrl(parameters: [
    'tableAction' => EditAction::getDefaultName(),
    'tableActionRecord' => $customer,
]); // /admin/customers?tableAction=edit&tableActionRecord=1
```

Or if you want to generate a URL for an action on the page like a `CreateAction` in the header, you can pass it in to the `action` parameter:

```
use App\Filament\Resources\CustomerResource;
use Filament\Actions\CreateAction;

CustomerResource::getUrl(parameters: [
    'action' => CreateAction::getDefaultName(),
]); // /admin/customers?action=create
```

Generating URLs to resources in other panels

If you have multiple panels in your app, `getUrl()` will generate a URL within the current panel. You can also indicate which panel the resource is associated with, by passing the panel ID to the `panel` argument:

```
use App\Filament\Resources\CustomerResource;

CustomerResource::getUrl(panel: 'marketing');
```

Customizing the resource Eloquent query

Within Filament, every query to your resource model will start with the `getEloquentQuery()` method.

Because of this, it's very easy to apply your own query constraints or [model scopes](#) that affect the entire resource:

```
public static function getEloquentQuery(): Builder
{
    return parent::getEloquentQuery() -> where('is_active', true);
}
```

Disabling global scopes

By default, Filament will observe all global scopes that are registered to your model. However, this may not be ideal if you wish to access, for example, soft deleted records.

To overcome this, you may override the `getEloquentQuery()` method that Filament uses:

```
public static function getEloquentQuery(): Builder
{
    return parent::getEloquentQuery() -> withoutGlobalScopes();
}
```

Alternatively, you may remove specific global scopes:

```
public static function getEloquentQuery(): Builder
{
    return parent::getEloquentQuery() ->withoutGlobalScopes([ActiveScope::class]);
}
```

More information about removing global scopes may be found in the [Laravel documentation](#).

Customizing the resource URL

By default, Filament will generate a URL based on the name of the resource. You can customize this by setting the `$slug` property on the resource:

```
protected static ?string $slug = 'pending-orders';
```

Resource sub-navigation

Sub-navigation allows the user to navigate between different pages within a resource. Typically, all pages in the sub-navigation will be related to the same record in the resource. For example, in a Customer resource, you may have a sub-navigation with the following pages:

- View customer, a `ViewRecord` page that provides a read-only view of the customer's details.
- Edit customer, an `EditRecord` page that allows the user to edit the customer's details.
- Edit customer contact, an `EditRecord` page that allows the user to edit the customer's contact details. You can [learn how to create more than one Edit page](#).
- Manage addresses, a `ManageRelatedRecords` page that allows the user to manage the customer's addresses.
- Manage payments, a `ManageRelatedRecords` page that allows the user to manage the customer's payments.

To add a sub-navigation to each "singular record" page in the resource, you can add the `getRecordSubNavigation()` method to the resource class:

```
use App\Filament\Resources\CustomerResource\Pages;
use Filament\Resources\Pages\Page;

public static function getRecordSubNavigation(Page $page): array
{
    return $page->generateNavigationItems([
        Pages\ViewCustomer::class,
        Pages>EditCustomer::class,
        Pages>EditCustomerContact::class,
        Pages\ManageCustomerAddresses::class,
        Pages\ManageCustomerPayments::class,
    ]);
}
```

Each item in the sub-navigation can be customized using the [same navigation methods as normal pages](#).

If you're looking to add sub-navigation to switch between entire resources and [custom pages](#), you might be looking for [clusters](#), which are used to group these together. The `getRecordSubNavigation()` method is intended to construct a navigation between pages that relate to a particular record inside a resource.

Sub-navigation position

The sub-navigation is rendered at the start of the page by default. You may change the position by setting the `$subNavigationPosition` property on the resource. The value may be `SubNavigationPosition::Start`,

`SubNavigationPosition::End`, or `SubNavigationPosition::Top` to render the sub-navigation as tabs:

```
use Filament\Pages\SubNavigationPosition;

protected static SubNavigationPosition $subNavigationPosition = SubNavigationPosition::End;
```

Deleting resource pages

If you'd like to delete a page from your resource, you can just delete the page file from the `Pages` directory of your resource, and its entry in the `getPages()` method.

For example, you may have a resource with records that may not be created by anyone. Delete the `Create` page file, and then remove it from `getPages()`:

```
public static function getPages(): array
{
    return [
        'index' => Pages\ListCustomers::route('/'),
        'edit' => Pages>EditCustomer::route('/{record}/edit'),
    ];
}
```

Deleting a page will not delete any actions that link to that page. Any actions will open a modal instead of sending the user to the non-existent page. For instance, the `CreateAction` on the List page, the `EditAction` on the table or View page, or the `ViewAction` on the table or Edit page. If you want to remove those buttons, you must delete the actions as well.

Listing Records

Using tabs to filter the records

You can add tabs above the table, which can be used to filter the records based on some predefined conditions. Each tab can scope the Eloquent query of the table in a different way. To register tabs, add a `getTabs()` method to the List page class, and return an array of `Tab` objects:

```
use Filament\Resources\Components\Tab;
use Illuminate\Database\Eloquent\Builder;

public function getTabs(): array
{
    return [
        'all' => Tab::make(),
        'active' => Tab::make()
            ->modifyQueryUsing(fn (Builder $query) => $query->where('active', true)),
        'inactive' => Tab::make()
            ->modifyQueryUsing(fn (Builder $query) => $query->where('active', false)),
    ];
}
```

Customizing the filter tab labels

The keys of the array will be used as identifiers for the tabs, so they can be persisted in the URL's query string. The label of each tab is also generated from the key, but you can override that by passing a label into the `make()` method of the tab:

```
use Filament\Resources\Components\Tab;
use Illuminate\Database\Eloquent\Builder;

public function getTabs(): array
{
    return [
        'all' => Tab::make('All customers'),
        'active' => Tab::make('Active customers')
            ->modifyQueryUsing(fn (Builder $query) => $query->where('active', true)),
        'inactive' => Tab::make('Inactive customers')
            ->modifyQueryUsing(fn (Builder $query) => $query->where('active', false)),
    ];
}
```

Adding icons to filter tabs

You can add icons to the tabs by passing an `icon` into the `icon()` method of the tab:

```
use Filament\Resources\Components\Tab;

Tab::make()
    ->icon('heroicon-m-user-group')
```

You can also change the icon's position to be after the label instead of before it, using the `iconPosition()` method:

```
use Filament\Support\Enums\IconPosition;

Tab::make()
->icon('heroicon-m-user-group')
->iconPosition(IconPosition::After)
```

Adding badges to filter tabs

You can add badges to the tabs by passing a string into the `badge()` method of the tab:

```
use Filament\Resources\Components\Tab;

Tab::make()
->badge(Customer::query()->where('active', true)->count())
```

Changing the color of filter tab badges

The color of a badge may be changed using the `badgeColor()` method:

```
use Filament\Resources\Components\Tab;

Tab::make()
->badge(Customer::query()->where('active', true)->count())
->badgeColor('success')
```

Adding extra attributes to filter tabs

You may also pass extra HTML attributes to filter tabs using `extraAttributes()`:

```
use Filament\Resources\Components\Tab;

Tab::make()
->extraAttributes(['data-cy' => 'statement-confirmed-tab'])
```

Customizing the default tab

To customize the default tab that is selected when the page is loaded, you can return the array key of the tab from the `getDefaultActiveTab()` method:

```

use Filament\Resources\Components\Tab;

public function getTabs(): array
{
    return [
        'all' => Tab::make(),
        'active' => Tab::make(),
        'inactive' => Tab::make(),
    ];
}

public function getDefaultActiveTab(): string | int | null
{
    return 'active';
}

```

Authorization

For authorization, Filament will observe any [model policies](#) that are registered in your app.

Users may access the List page if the `viewAny()` method of the model policy returns `true`.

The `reorder()` method is used to control [reordering a record](#).

Customizing the table Eloquent query

Although you can [customize the Eloquent query for the entire resource](#), you may also make specific modifications for the List page table. To do this, use the `modifyQueryUsing()` method in the `table()` method of the resource:

```

use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;

public static function table(Table $table): Table
{
    return $table
        ->modifyQueryUsing(fn (Builder $query) => $query->withoutGlobalScopes());
}

```

Custom list page view

For further customization opportunities, you can override the static `$view` property on the page class to a custom view in your app:

```
protected static string $view = 'filament.resources.users.pages.list-users';
```

This assumes that you have created a view at `resources/views/filament/resources/users/pages/list-users.blade.php`.

Here's a basic example of what that view might contain:

```

<x-filament-panels::page>
    {{ $this->table }}
</x-filament-panels::page>

```

To see everything that the default view contains, you can check the

`vendor/filament/filament/resources/views/resources/pages/list-records.blade.php` file in your project.

Creating Records

Customizing data before saving

Sometimes, you may wish to modify form data before it is finally saved to the database. To do this, you may define a `mutateFormDataBeforeCreate()` method on the Create page class, which accepts the `$data` as an array, and returns the modified version:

```
protected function mutateFormDataBeforeCreate(array $data): array
{
    $data['user_id'] = auth()->id();

    return $data;
}
```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

Customizing the creation process

You can tweak how the record is created using the `handleRecordCreation()` method on the Create page class:

```
use Illuminate\Database\Eloquent\Model;

protected function handleRecordCreation(array $data): Model
{
    return static::getModel()::create($data);
}
```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

Customizing redirects

By default, after saving the form, the user will be redirected to the [Edit page](#) of the resource, or the [View page](#) if it is present.

You may set up a custom redirect when the form is saved by overriding the `getRedirectUrl()` method on the Create page class.

For example, the form can redirect back to the [List page](#):

```
protected function getRedirectUrl(): string
{
    return $this->getResource()::getUrl('index');
}
```

If you wish to be redirected to the previous page, else the index page:

```
protected function getRedirectUrl(): string
{
    return $this->previousUrl ?? $this->getResource()::getUrl('index');
}
```

Customizing the save notification

When the record is successfully created, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, define a `getCreatedNotificationTitle()` method on the create page class:

```
protected function getCreatedNotificationTitle(): ?string
{
    return 'User registered';
}
```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

You may customize the entire notification by overriding the `getCreatedNotification()` method on the create page class:

```
use Filament\Notifications\Notification;

protected function getCreatedNotification(): ?Notification
{
    return Notification::make()
        ->success()
        ->title('User registered')
        ->body('The user has been created successfully.');
}
```

To disable the notification altogether, return `null` from the `getCreatedNotification()` method on the create page class:

```
use Filament\Notifications\Notification;

protected function getCreatedNotification(): ?Notification
{
    return null;
}
```

Lifecycle hooks

Hooks may be used to execute code at various points within a page's lifecycle, like before a form is saved. To set up a hook, create a protected method on the Create page class with the name of the hook:

```
protected function beforeCreate(): void
{
    // ...
}
```

In this example, the code in the `beforeCreate()` method will be called before the data in the form is saved to the database.

There are several available hooks for the Create page:

```
use Filament\Resources\Pages\CreateRecord;

class CreateUser extends CreateRecord
{
    // ...

    protected function beforeFill(): void
    {
        // Runs before the form fields are populated with their default values.
    }

    protected function afterFill(): void
    {
        // Runs after the form fields are populated with their default values.
    }

    protected function beforeValidate(): void
    {
        // Runs before the form fields are validated when the form is submitted.
    }

    protected function afterValidate(): void
    {
        // Runs after the form fields are validated when the form is submitted.
    }

    protected function beforeCreate(): void
    {
        // Runs before the form fields are saved to the database.
    }

    protected function afterCreate(): void
    {
        // Runs after the form fields are saved to the database.
    }
}
```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

Halting the creation process

At any time, you may call `$this->halt()` from inside a lifecycle hook or mutation method, which will halt the entire creation process:

```

use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

protected function beforeCreate(): void
{
    if (! auth()->user()->team->subscribed()) {
        Notification::make()
            ->warning()
            ->title('You don\'t have an active subscription!')
            ->body('Choose a plan to continue.')
            ->persistent()
            ->actions([
                Action::make('subscribe')
                    ->button()
                    ->url(route('subscribe'), shouldOpenInNewTab: true),
            ])
            ->send();
    }

    $this->halt();
}
}

```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

Authorization

For authorization, Filament will observe any [model policies](#) that are registered in your app.

Users may access the Create page if the `create()` method of the model policy returns `true`.

Using a wizard

You may easily transform the creation process into a multistep wizard.

On the page class, add the corresponding `HasWizard` trait:

```

use App\Filament\Resources\CategoryResource;
use Filament\Resources\Pages>CreateRecord;

class CreateCategory extends CreateRecord
{
    use CreateRecord\Concerns\HasWizard;

    protected static string $resource = CategoryResource::class;

    protected function getSteps(): array
    {
        return [
            // ...
        ];
    }
}

```

Inside the `getSteps()` array, return your [wizard steps](#):

```

use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Toggle;
use Filament\Forms\Components\Wizard\Step;

protected function getSteps(): array
{
    return [
        Step::make('Name')
            ->description('Give the category a clear and unique name')
            ->schema([
                TextInput::make('name')
                    ->required()
                    ->live()
                    ->afterStateUpdated(fn ($state, callable $set) => $set('slug',
Str::slug($state))),
                TextInput::make('slug')
                    ->disabled()
                    ->required()
                    ->unique(Category::class, 'slug', fn ($record) => $record),
            ]),
        Step::make('Description')
            ->description('Add some extra details')
            ->schema([
                MarkdownEditor::make('description')
                    ->columnSpan('full'),
            ]),
        Step::make('Visibility')
            ->description('Control who can view it')
            ->schema([
                Toggle::make('is_visible')
                    ->label('Visible to customers.')
                    ->default(true),
            ]),
    ];
}

```

Alternatively, if you're creating records in a modal action, check out the [Actions documentation](#).

Now, create a new record to see your wizard in action! Edit will still use the form defined within the resource class.

If you'd like to allow free navigation, so all the steps are skippable, override the `hasSkippableSteps()` method:

```

public function hasSkippableSteps(): bool
{
    return true;
}

```

Sharing fields between the resource form and wizards

If you'd like to reduce the amount of repetition between the resource form and wizard steps, it's a good idea to extract public static resource functions for your fields, where you can easily retrieve an instance of a field from the resource or the wizard:

```
use Filament\Forms;
use Filament\Forms\Form;
use Filament\Resources\Resource;

class CategoryResource extends Resource
{
    public static function form(Form $form): Form
    {
        return $form
            ->schema([
                static::getNameFormField(),
                static::getSlugFormField(),
                // ...
            ]);
    }

    public static function getNameFormField(): Forms\Components\TextInput
    {
        return TextInput::make('name')
            ->required()
            ->live()
            ->afterStateUpdated(fn ($state, callable $set) => $set('slug', Str::slug($state)));
    }

    public static function getSlugFormField(): Forms\Components\TextInput
    {
        return TextInput::make('slug')
            ->disabled()
            ->required()
            ->unique(Category::class, 'slug', fn ($record) => $record);
    }
}
```

```

use App\Filament\Resources\CategoryResource;
use Filament\Resources\Pages>CreateRecord;

class CreateCategory extends CreateRecord
{
    use CreateRecord\Concerns\HasWizard;

    protected static string $resource = CategoryResource::class;

    protected function getSteps(): array
    {
        return [
            Step::make('Name')
                ->description('Give the category a clear and unique name')
                ->schema([
                    CategoryResource::getNameFormField(),
                    CategoryResource::getSlugFormField(),
                ]),
            // ...
        ];
    }
}

```

Importing resource records

Filament includes an `ImportAction` that you can add to the `getHeaderActions()` of the [List page](#). It allows users to upload a CSV of data to import into the resource:

```

use App\Filament\Imports\ProductImporter;
use Filament\Actions;

protected function getHeaderActions(): array
{
    return [
        Actions\ImportAction::make()
            ->importer(ProductImporter::class),
        Actions\CreateAction::make(),
    ];
}

```

The "importer" class [needs to be created](#) to tell Filament how to import each row of the CSV. You can learn everything about the `ImportAction` in the [Actions documentation](#).

Custom actions

"Actions" are buttons that are displayed on pages, which allow the user to run a Livewire method on the page or visit a URL.

On resource pages, actions are usually in 2 places: in the top right of the page, and below the form.

For example, you may add a new button action in the header of the Create page:

```

use App\Filament\Imports\UserImporter;
use Filament\Actions;
use Filament\Resources\Pages\CreateRecord;

class CreateUser extends CreateRecord
{
    // ...

    protected function getHeaderActions(): array
    {
        return [
            Actions\ImportAction::make()
                ->importer(UserImporter::class),
        ];
    }
}

```

Or, a new button next to "Create" below the form:

```

use Filament\Actions\Action;
use Filament\Resources\Pages\CreateRecord;

class CreateUser extends CreateRecord
{
    // ...

    protected function getFormActions(): array
    {
        return [
            ...parent::getFormActions(),
            Action::make('close')->action('createAndClose'),
        ];
    }

    public function createAndClose(): void
    {
        // ...
    }
}

```

To view the entire actions API, please visit the [pages section](#).

Adding a create action button to the header

The "Create" button can be moved to the header of the page by overriding the `getHeaderActions()` method and using `getCreateFormAction()`. You need to pass `formId()` to the action, to specify that the action should submit the form with the ID of `form`, which is the `<form>` ID used in the view of the page:

```
protected function getHeaderActions(): array
{
    return [
        $this->getCreateFormAction()
            ->formId('form'),
    ];
}
```

You may remove all actions from the form by overriding the `getFormActions()` method to return an empty array:

```
protected function getFormActions(): array
{
    return [];
}
```

Custom view

For further customization opportunities, you can override the static `$view` property on the page class to a custom view in your app:

```
protected static string $view = 'filament.resources.users.pages.create-user';
```

This assumes that you have created a view at `resources/views/filament/resources/users/pages/create-user.blade.php`.

Here's a basic example of what that view might contain:

```
<x-filament-panels::page>
<x-filament-panels::form wire:submit="create">
    {{ $this->form }}

    <x-filament-panels::form.actions
        :actions="$this->getCachedFormActions()"
        :full-width="$this->hasFullWidthFormActions()" />
</x-filament-panels::form>
</x-filament-panels::page>
```

To see everything that the default view contains, you can check the

`vendor/filament/filament/resources/views/resources/pages/create-record.blade.php` file in your project.

Editing Records

Customizing data before filling the form

You may wish to modify the data from a record before it is filled into the form. To do this, you may define a `mutateFormDataBeforeFill()` method on the Edit page class to modify the `$data` array, and return the modified version before it is filled into the form:

```
protected function mutateFormDataBeforeFill(array $data): array
{
    $data['user_id'] = auth()->id();

    return $data;
}
```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

Customizing data before saving

Sometimes, you may wish to modify form data before it is finally saved to the database. To do this, you may define a `mutateFormDataBeforeSave()` method on the Edit page class, which accepts the `$data` as an array, and returns it modified:

```
protected function mutateFormDataBeforeSave(array $data): array
{
    $data['last_edited_by_id'] = auth()->id();

    return $data;
}
```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

Customizing the saving process

You can tweak how the record is updated using the `handleRecordUpdate()` method on the Edit page class:

```
use Illuminate\Database\Eloquent\Model;

protected function handleRecordUpdate(Model $record, array $data): Model
{
    $record->update($data);

    return $record;
}
```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

Customizing redirects

By default, saving the form will not redirect the user to another page.

You may set up a custom redirect when the form is saved by overriding the `getRedirectUrl()` method on the Edit page class.

For example, the form can redirect back to the [List page](#) of the resource:

```
protected function getRedirectUrl(): string
{
    return $this->getResource()::getUrl('index');
}
```

Or the [View page](#):

```
protected function getRedirectUrl(): string
{
    return $this->getResource()::getUrl('view', ['record' => $this->getRecord()]);
}
```

If you wish to be redirected to the previous page, else the index page:

```
protected function getRedirectUrl(): string
{
    return $this->previousUrl ?? $this->getResource()::getUrl('index');
}
```

Customizing the save notification

When the record is successfully updated, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, define a `getSavedNotificationTitle()` method on the edit page class:

```
protected function getSavedNotificationTitle(): ?string
{
    return 'User updated';
}
```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

You may customize the entire notification by overriding the `getSavedNotification()` method on the edit page class:

```
use Filament\Notifications\Notification;

protected function getSavedNotification(): ?Notification
{
    return Notification::make()
        ->success()
        ->title('User updated')
        ->body('The user has been saved successfully.');
}
```

To disable the notification altogether, return `null` from the `getSavedNotification()` method on the edit page class:

```
use Filament\Notifications\Notification;

protected function getSavedNotification(): ?Notification
{
    return null;
}
```

Lifecycle hooks

Hooks may be used to execute code at various points within a page's lifecycle, like before a form is saved. To set up a hook, create a protected method on the Edit page class with the name of the hook:

```
protected function beforeSave(): void
{
    // ...
}
```

In this example, the code in the `beforeSave()` method will be called before the data in the form is saved to the database.

There are several available hooks for the Edit pages:

```

use Filament\Resources\Pages>EditRecord;

class EditUser extends EditRecord
{
    // ...

    protected function beforeFill(): void
    {
        // Runs before the form fields are populated from the database.
    }

    protected function afterFill(): void
    {
        // Runs after the form fields are populated from the database.
    }

    protected function beforeValidate(): void
    {
        // Runs before the form fields are validated when the form is saved.
    }

    protected function afterValidate(): void
    {
        // Runs after the form fields are validated when the form is saved.
    }

    protected function beforeSave(): void
    {
        // Runs before the form fields are saved to the database.
    }

    protected function afterSave(): void
    {
        // Runs after the form fields are saved to the database.
    }
}

```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

Saving a part of the form independently

You may want to allow the user to save a part of the form independently of the rest of the form. One way to do this is with a [section action in the header or footer](#). From the `action()` method, you can call `saveFormComponentOnly()`, passing in the `Section` component that you want to save:

```

use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Section;
use Filament\Notifications\Notification;
use Filament\Resources\Pages>EditRecord;

Section::make('Rate limiting')
    ->schema([
        // ...
    ])
    ->footerActions([
        fn (string $operation): Action => Action::make('save')
            ->action(function (Section $component, EditRecord $livewire) {
                $livewire->saveFormComponentOnly($component);

                Notification::make()
                    ->title('Rate limiting saved')
                    ->body('The rate limiting settings have been saved successfully.')
                    ->success()
                    ->send();
            })
            ->visible($operation === 'edit'),
    ])
)

```

The `$operation` helper is available, to ensure that the action is only visible when the form is being edited.

Halting the saving process

At any time, you may call `$this->halt()` from inside a lifecycle hook or mutation method, which will halt the entire saving process:

```

use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

protected function beforeSave(): void
{
    if (! $this->getRecord()->team->subscribed()) {
        Notification::make()
            ->warning()
            ->title('You don\'t have an active subscription!')
            ->body('Choose a plan to continue.')
            ->persistent()
            ->actions([
                Action::make('subscribe')
                    ->button()
                    ->url(route('subscribe'), shouldOpenInNewTab: true),
            ])
            ->send();

        $this->halt();
    }
}

```

Alternatively, if you're editing records in a modal action, check out the [Actions documentation](#).

Authorization

For authorization, Filament will observe any model policies that are registered in your app.

Users may access the Edit page if the `update()` method of the model policy returns `true`.

They also have the ability to delete the record if the `delete()` method of the policy returns `true`.

Custom actions

"Actions" are buttons that are displayed on pages, which allow the user to run a Livewire method on the page or visit a URL.

On resource pages, actions are usually in 2 places: in the top right of the page, and below the form.

For example, you may add a new button action next to "Delete" on the Edit page:

```
use Filament\Actions;
use Filament\Resources\Pages>EditRecord;

class EditUser extends EditRecord
{
    // ...

    protected function getHeaderActions(): array
    {
        return [
            Actions\Action::make('impersonate')
                ->action(function (): void {
                    // ...
                }),
            Actions\DeleteAction::make(),
        ];
    }
}
```

Or, a new button next to "Save" below the form:

```

use Filament\Actions\Action;
use Filament\Resources\Pages>EditRecord;

class EditUser extends EditRecord
{
    // ...

    protected function getFormActions(): array
    {
        return [
            ...parent::getFormActions(),
            Action::make('close')->action('saveAndClose'),
        ];
    }

    public function saveAndClose(): void
    {
        // ...
    }
}

```

To view the entire actions API, please visit the [pages section](#).

Adding a save action button to the header

The "Save" button can be moved to the header of the page by overriding the `getHeaderActions()` method and using `getSaveFormAction()`. You need to pass `formId()` to the action, to specify that the action should submit the form with the ID of `form`, which is the `<form>` ID used in the view of the page:

```

protected function getHeaderActions(): array
{
    return [
        $this->getSaveFormAction()
            ->formId('form'),
    ];
}

```

You may remove all actions from the form by overriding the `getFormActions()` method to return an empty array:

```

protected function getFormActions(): array
{
    return [];
}

```

Creating another Edit page

One Edit page may not be enough space to allow users to navigate many form fields. You can create as many Edit pages for a resource as you want. This is especially useful if you are using [resource sub-navigation](#), as you are then easily able to switch between the different Edit pages.

To create an Edit page, you should use the `make:filament-page` command:

```
php artisan make:filament-page EditCustomerContact --resource=CustomerResource --type>EditRecord
```

You must register this new page in your resource's `getPages()` method:

```
public static function getPages(): array
{
    return [
        'index' => Pages\ListCustomers::route('/'),
        'create' => Pages\CreateCustomer::route('/create'),
        'view' => Pages\ViewCustomer::route('/{record}'),
        'edit' => Pages>EditCustomer::route('/{record}/edit'),
        'edit-contact' => Pages>EditCustomerContact::route('/{record}/edit/contact'),
    ];
}
```

Now, you can define the `form()` for this page, which can contain other fields that are not present on the main Edit page:

```
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            // ...
        ]);
}
```

Adding edit pages to resource sub-navigation

If you're using resource sub-navigation, you can register this page as normal in `getRecordSubNavigation()` of the resource:

```
use App\Filament\Resources\CustomerResource\Pages;
use Filament\Resources\Pages\Page;

public static function getRecordSubNavigation(Page $page): array
{
    return $page->generateNavigationItems([
        // ...
        Pages>EditCustomerContact::class,
    ]);
}
```

Custom views

For further customization opportunities, you can override the static `$view` property on the page class to a custom view in your app:

```
protected static string $view = 'filament.resources.users.pages.edit-user';
```

This assumes that you have created a view at `resources/views/filament/resources/users/pages/edit-user.blade.php`.

Here's a basic example of what that view might contain:

```
<x-filament-panels::page>
  <x-filament-panels::form wire:submit="save">
    {{ $this->form }}

    <x-filament-panels::form.actions
      :actions="$this->getCachedFormActions()"
      :full-width="$this->hasFullWidthFormActions()"
    />
  </x-filament-panels::form>

  @if (count($relationManagers = $this->getRelationManagers()))
    <x-filament-panels::resources.relation-managers
      :active-manager="$this->activeRelationManager"
      :managers="$relationManagers"
      :owner-record="$record"
      :page-class="static::class"
    />
  @endif
</x-filament-panels::page>
```

To see everything that the default view contains, you can check the

`vendor/filament/filament/resources/views/resources/pages/edit-record.blade.php` file in your project.

Viewing Records

Creating a resource with a View page

To create a new resource with a View page, you can use the `--view` flag:

```
php artisan make:filament-resource User --view
```

Using an infolist instead of a disabled form

By default, the View page will display a disabled form with the record's data. If you preferred to display the record's data in an "infolist", you can define an `infolist()` method on the resource class:

```
use Filament\Infolists;
use Filament\Infolists\Infolist;

public static function infolist(Infolist $infolist): Infolist
{
    return $infolist
        ->schema([
            Infolists\Components\TextEntry::make('name'),
            Infolists\Components\TextEntry::make('email'),
            Infolists\Components\TextEntry::make('notes')
                ->columnSpanFull(),
        ]);
}
```

The `schema()` method is used to define the structure of your infolist. It is an array of [entries](#) and [layout components](#), in the order they should appear in your infolist.

Check out the Infolists docs for a [guide](#) on how to build infolists with Filament.

Adding a View page to an existing resource

If you want to add a View page to an existing resource, create a new page in your resource's `Pages` directory:

```
php artisan make:filament-page ViewUser --resource=UserResource --type=ViewRecord
```

You must register this new page in your resource's `getPages()` method:

```
public static function getPages(): array
{
    return [
        'index' => Pages\ListUsers::route('/'),
        'create' => Pages\CreateUser::route('/create'),
        'view' => Pages\ViewUser::route('/{record}'),
        'edit' => Pages>EditUser::route('/{record}/edit'),
    ];
}
```

Viewing records in modals

If your resource is simple, you may wish to view records in modals rather than on the [View page](#). If this is the case, you can just [delete the view page](#).

If your resource doesn't contain a `ViewAction`, you can add one to the `$table->actions()` array:

```
use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->actions([
            Tables\Actions\ViewAction::make(),
            // ...
        ]);
}
```

Customizing data before filling the form

You may wish to modify the data from a record before it is filled into the form. To do this, you may define a `mutateFormDataBeforeFill()` method on the View page class to modify the `$data` array, and return the modified version before it is filled into the form:

```
protected function mutateFormDataBeforeFill(array $data): array
{
    $data['user_id'] = auth()->id();

    return $data;
}
```

Alternatively, if you're viewing records in a modal action, check out the [Actions documentation](#).

Authorization

For authorization, Filament will observe any [model policies](#) that are registered in your app.

Users may access the View page if the `view()` method of the model policy returns `true`.

Creating another View page

One View page may not be enough space to allow users to navigate a lot of information. You can create as many View pages for a resource as you want. This is especially useful if you are using [resource sub-navigation](#), as you are then easily able to switch between the different View pages.

To create a View page, you should use the `make:filament-page` command:

```
php artisan make:filament-page ViewCustomerContact --resource=CustomerResource --type=ViewRecord
```

You must register this new page in your resource's `getPages()` method:

```
public static function getPages(): array
{
    return [
        'index' => Pages\ListCustomers::route('/'),
        'create' => Pages\CreateCustomer::route('/create'),
        'view' => Pages\ViewCustomer::route('/{record}'),
        'view-contact' => Pages\ViewCustomerContact::route('/{record}/contact'),
        'edit' => Pages>EditCustomer::route('/{record}/edit'),
    ];
}
```

Now, you can define the `infolist()` or `form()` for this page, which can contain other components that are not present on the main View page:

```
use Filament\Infolists\Infolist;

public function infolist(Infolist $infolist): Infolist
{
    return $infolist
        ->schema([
            // ...
        ]);
}
```

Adding view pages to resource sub-navigation

If you're using resource sub-navigation, you can register this page as normal in `getRecordSubNavigation()` of the resource:

```
use App\Filament\Resources\CustomerResource\Pages;
use Filament\Resources\Pages\Page;

public static function getRecordSubNavigation(Page $page): array
{
    return $page->generateNavigationItems([
        // ...
        Pages\ViewCustomerContact::class,
    ]);
}
```

Custom view

For further customization opportunities, you can override the static `$view` property on the page class to a custom view in your app:

```
protected static string $view = 'filament.resources.users.pages.view-user';
```

This assumes that you have created a view at `resources/views/filament/resources/users/pages/view-user.blade.php`.

Here's a basic example of what that view might contain:

```
<x-filament-panels::page>
@if ($this->hasInfolist())
    {{ $this->infolist }}
@else
    {{ $this->form }}
@endif

@if (count($relationManagers = $this->getRelationManagers()))
<x-filament-panels::resources.relation-managers
:active-manager="$this->activeRelationManager"
:managers="$relationManagers"
:owner-record="$record"
:page-class="static::class"
/>
@endif
</x-filament-panels::page>
```

To see everything that the default view contains, you can check the

`vendor/filament/filament/resources/views/resources/pages/view-record.blade.php` file in your project.

Deleting Records

Handling soft deletes

Creating a resource with soft delete

By default, you will not be able to interact with deleted records in the app. If you'd like to add functionality to restore, force delete and filter trashed records in your resource, use the `--soft-deletes` flag when generating the resource:

```
php artisan make:filament-resource Customer --soft-deletes
```

Adding soft deletes to an existing resource

Alternatively, you may add soft deleting functionality to an existing resource.

Firstly, you must update the resource:

```

use Filament\Tables;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\SoftDeletingScope;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters([
            Tables\Filters\TrashedFilter::make(),
            // ...
        ])
        ->actions([
            // You may add these actions to your table if you're using a simple
            // resource, or you just want to be able to delete records without
            // leaving the table.
            Tables\Actions\DeleteAction::make(),
            Tables\Actions\ForceDeleteAction::make(),
            Tables\Actions\RestoreAction::make(),
            // ...
        ])
        ->bulkActions([
            Tables\Actions\BulkActionGroup::make([
                Tables\Actions\DeleteBulkAction::make(),
                Tables\Actions\ForceDeleteBulkAction::make(),
                Tables\Actions\RestoreBulkAction::make(),
                // ...
            ]),
        ]);
}

public static function getEloquentQuery(): Builder
{
    return parent::getEloquentQuery()
        ->withoutGlobalScopes([
            SoftDeletingScope::class,
        ]);
}

```

Now, update the Edit page class if you have one:

```
use Filament\Actions;

protected function getHeaderActions(): array
{
    return [
        Actions\DeleteAction::make(),
        Actions\ForceDeleteAction::make(),
        Actions\RestoreAction::make(),
        // ...
    ];
}
```

Deleting records on the List page

By default, you can bulk-delete records in your table. You may also wish to delete single records, using a `DeleteAction`:

```
use Filament\Tables;
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->actions([
            // ...
            Tables\Actions\DeleteAction::make(),
        ]);
}
```

Authorization

For authorization, Filament will observe any [model policies](#) that are registered in your app.

Users may delete records if the `delete()` method of the model policy returns `true`.

They also have the ability to bulk-delete records if the `deleteAny()` method of the policy returns `true`. Filament uses the `deleteAny()` method because iterating through multiple records and checking the `delete()` policy is not very performant.

Authorizing soft deletes

The `forceDelete()` policy method is used to prevent a single soft-deleted record from being force-deleted.

`forceDeleteAny()` is used to prevent records from being bulk force-deleted. Filament uses the `forceDeleteAny()` method because iterating through multiple records and checking the `forceDelete()` policy is not very performant.

The `restore()` policy method is used to prevent a single soft-deleted record from being restored. `restoreAny()` is used to prevent records from being bulk restored. Filament uses the `restoreAny()` method because iterating through multiple records and checking the `restore()` policy is not very performant.

Relation Managers

Choosing the right tool for the job

Filament provides many ways to manage relationships in the app. Which feature you should use depends on the type of relationship you are managing, and which UI you are looking for.

Relation managers - interactive tables underneath your resource forms

These are compatible with `HasMany`, `HasManyThrough`, `BelongsToMany`, `MorphMany` and `MorphToMany` relationships.

Relation managers are interactive tables that allow administrators to list, create, attach, associate, edit, detach, dissociate and delete related records without leaving the resource's Edit or View page.

Select & checkbox list - choose from existing records or create a new one

These are compatible with `BelongsTo`, `MorphTo` and `BelongsToMany` relationships.

Using a select, users will be able to choose from a list of existing records. You may also add a button that allows you to create a new record inside a modal, without leaving the page.

When using a `BelongsToMany` relationship with a select, you'll be able to select multiple options, not just one. Records will be automatically added to your pivot table when you submit the form. If you wish, you can swap out the multi-select dropdown with a simple checkbox list. Both components work in the same way.

Repeaters - CRUD multiple related records inside the owner's form

These are compatible with `HasMany` and `MorphMany` relationships.

Repeaters are standard form components, which can render a repeatable set of fields infinitely. They can be hooked up to a relationship, so records are automatically read, created, updated, and deleted from the related table. They live inside the main form schema, and can be used inside resource pages, as well as nesting within action modals.

From a UX perspective, this solution is only suitable if your related model only has a few fields. Otherwise, the form can get very long.

Layout form components - saving form fields to a single relationship

These are compatible with `BelongsTo`, `hasOne` and `MorphOne` relationships.

All layout form components (Grid, Section, Fieldset, etc.) have a `relationship()` method. When you use this, all fields within that layout are saved to the related model instead of the owner's model:

```
use Filament\Forms\Components\Fieldset;
use Filament\Forms\Components\FileUpload;
use Filament\Forms\Components\Textarea;
use Filament\Forms\Components\TextInput;

Fieldset::make('Metadata')
    ->relationship('metadata')
    ->schema([
        TextInput::make('title'),
        Textarea::make('description'),
        FileUpload::make('image'),
    ])
)
```

In this example, the `title`, `description` and `image` are automatically loaded from the `metadata` relationship, and saved again when the form is submitted. If the `metadata` record does not exist, it is automatically created.

This feature is explained more in depth in the [Forms documentation](#). Please visit that page for more information about how to use it.

Creating a relation manager

To create a relation manager, you can use the `make:filament-relation-manager` command:

```
php artisan make:filament-relation-manager CategoryResource posts title
```

- `CategoryResource` is the name of the resource class for the owner (parent) model.
- `posts` is the name of the relationship you want to manage.
- `title` is the name of the attribute that will be used to identify posts.

This will create a `CategoryResource/RelationManagers/PostsRelationManager.php` file. This contains a class where you are able to define a `form` and `table` for your relation manager:

```

use Filament\Forms;
use Filament\Forms\Form;
use Filament\Tables;
use Filament\Tables\Table;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('title')->required(),
            // ...
        ]);
}

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('title'),
            // ...
        ]);
}

```

You must register the new relation manager in your resource's `getRelations()` method:

```

public static function getRelations(): array
{
    return [
        RelationManagers\PostsRelationManager::class,
    ];
}

```

Once a table and form have been defined for the relation manager, visit the [Edit](#) or [View](#) page of your resource to see it in action.

Read-only mode

Relation managers are usually displayed on either the Edit or View page of a resource. On the View page, Filament will automatically hide all actions that modify the relationship, such as create, edit, and delete. We call this "read-only mode", and it is there by default to preserve the read-only behavior of the View page. However, you can disable this behavior, by overriding the `isReadOnly()` method on the relation manager class to return `false` all the time:

```

public function isReadOnly(): bool
{
    return false;
}

```

Alternatively, if you hate this functionality, you can disable it for all relation managers at once in the panel [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->readOnlyRelationManagersOnResourceViewPagesByDefault(false);
}
```

Unconventional inverse relationship names

For inverse relationships that do not follow Laravel's naming guidelines, you may wish to use the `inverseRelationship()` method on the table:

```
use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('title'),
            // ...
        ])
        ->inverseRelationship('section'); // Since the inverse related model is `Category`, this
        is normally `category`, not `section`.
}
```

Handling soft deletes

By default, you will not be able to interact with deleted records in the relation manager. If you'd like to add functionality to restore, force delete and filter trashed records in your relation manager, use the `--soft-deletes` flag when generating the relation manager:

```
php artisan make:filament-relation-manager CategoryResource posts title --soft-deletes
```

You can find out more about soft deleting [here](#).

Listing related records

Related records will be listed in a table. The entire relation manager is based around this table, which contains actions to [create](#), [edit](#), [attach / detach](#), [associate / dissociate](#), and delete records.

You may use any features of the [Table Builder](#) to customize relation managers.

Listing with pivot attributes

For `BelongsToMany` and `MorphToMany` relationships, you may also add pivot table attributes. For example, if you have a `TeamsRelationManager` for your `UserResource`, and you want to add the `role` pivot attribute to the table, you can use:

```
use Filament\Tables;

public function table(Table $table): Table
{
    return $table
        ->columns([
            Tables\Columns\TextColumn::make('name'),
            Tables\Columns\TextColumn::make('role'),
        ]);
}
```

Please ensure that any pivot attributes are listed in the `withPivot()` method of the relationship *and* inverse relationship.

Creating related records

Creating with pivot attributes

For `BelongsToMany` and `MorphToMany` relationships, you may also add pivot table attributes. For example, if you have a `TeamsRelationManager` for your `UserResource`, and you want to add the `role` pivot attribute to the create form, you can use:

```
use Filament\Forms;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')->required(),
            Forms\Components\TextInput::make('role')->required(),
            // ...
        ]);
}
```

Please ensure that any pivot attributes are listed in the `withPivot()` method of the relationship *and* inverse relationship.

Customizing the `CreateAction`

To learn how to customize the `CreateAction`, including mutating the form data, changing the notification, and adding lifecycle hooks, please see the [Actions documentation](#).

Editing related records

Editing with pivot attributes

For `BelongsToMany` and `MorphToMany` relationships, you may also edit pivot table attributes. For example, if you have a `TeamsRelationManager` for your `UserResource`, and you want to add the `role` pivot attribute to the edit form, you can use:

```

use Filament\Forms;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')->required(),
            Forms\Components\TextInput::make('role')->required(),
            // ...
        ]);
}

```

Please ensure that any pivot attributes are listed in the `withPivot()` method of the relationship *and* inverse relationship.

Customizing the `EditAction`

To learn how to customize the `EditAction`, including mutating the form data, changing the notification, and adding lifecycle hooks, please see the [Actions documentation](#).

Attaching and detaching records

Filament is able to attach and detach records for `BelongsToMany` and `MorphToMany` relationships.

When generating your relation manager, you may pass the `--attach` flag to also add `AttachAction`, `DetachAction` and `DetachBulkAction` to the table:

```
php artisan make:filament-relation-manager CategoryResource posts title --attach
```

Alternatively, if you've already generated your resource, you can just add the actions to the `$table` arrays:

```

use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->headerActions([
            // ...
            Tables\Actions\AttachAction::make(),
        ])
        ->actions([
            // ...
            Tables\Actions\DetachAction::make(),
        ])
        ->bulkActions([
            Tables\Actions\BulkActionGroup::make([
                // ...
                Tables\Actions\DetachBulkAction::make(),
            ]),
        ]);
}

```

Preloading the attachment modal select options

By default, as you search for a record to attach, options will load from the database via AJAX. If you wish to preload these options when the form is first loaded instead, you can use the `preloadRecordSelect()` method of `AttachAction`:

```

use Filament\Tables\Actions\AttachAction;

AttachAction::make()
    ->preloadRecordSelect()

```

Attaching with pivot attributes

When you attach record with the `Attach` button, you may wish to define a custom form to add pivot attributes to the relationship:

```

use Filament\Forms;
use Filament\Tables\Actions\AttachAction;

AttachAction::make()
    ->form(fn (AttachAction $action): array => [
        $action->getRecordSelect(),
        Forms\Components\TextInput::make('role')->required(),
    ])

```

In this example, `$action->getRecordSelect()` returns the select field to pick the record to attach. The `role` text input is then saved to the pivot table's `role` column.

Please ensure that any pivot attributes are listed in the `withPivot()` method of the relationship and inverse relationship.

Scoping the options to attach

You may want to scope the options available to `AttachAction`:

```
use Filament\Tables\Actions\AttachAction;
use Illuminate\Database\Eloquent\Builder;

AttachAction::make()
->recordSelectOptionsQuery(fn (Builder $query) => $query->whereBelongsTo(auth()->user()) )
```

Searching the options to attach across multiple columns

By default, the options available to `AttachAction` will be searched in the `recordTitleAttribute()` of the table. If you wish to search across multiple columns, you can use the `recordSelectSearchColumns()` method:

```
use Filament\Tables\Actions\AttachAction;

AttachAction::make()
->recordSelectSearchColumns(['title', 'description'])
```

Attaching multiple records

The `multiple()` method on the `AttachAction` component allows you to select multiple values:

```
use Filament\Tables\Actions\AttachAction;

AttachAction::make()
->multiple()
```

Customizing the select field in the attached modal

You may customize the select field object that is used during attachment by passing a function to the `recordSelect()` method:

```
use Filament\Forms\Components>Select;
use Filament\Tables\Actions\AttachAction;

AttachAction::make()
->recordSelect(
    fn (Select $select) => $select->placeholder('Select a post'),
)
```

Handling duplicates

By default, you will not be allowed to attach a record more than once. This is because you must also set up a primary `id` column on the pivot table for this feature to work.

Please ensure that the `id` attribute is listed in the `withPivot()` method of the relationship *and* inverse relationship.

Finally, add the `allowDuplicates()` method to the table:

```
public function table(Table $table): Table
{
    return $table
        ->allowDuplicates();
}
```

Associating and dissociating records

Filament is able to associate and dissociate records for `HasMany` and `MorphMany` relationships.

When generating your relation manager, you may pass the `--associate` flag to also add `AssociateAction`, `DissociateAction` and `DissociateBulkAction` to the table:

```
php artisan make:filament-relation-manager CategoryResource posts title --associate
```

Alternatively, if you've already generated your resource, you can just add the actions to the `$table` arrays:

```
use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->headerActions([
            // ...
            Tables\Actions\AssociateAction::make(),
        ])
        ->actions([
            // ...
            Tables\Actions\DissociateAction::make(),
        ])
        ->bulkActions([
            Tables\Actions\BulkActionGroup::make([
                // ...
                Tables\Actions\DissociateBulkAction::make(),
            ]),
        ]);
}
```

Preloading the associate modal select options

By default, as you search for a record to associate, options will load from the database via AJAX. If you wish to preload these options when the form is first loaded instead, you can use the `preloadRecordSelect()` method of `AssociateAction`:

```
use Filament\Tables\Actions\AssociateAction;

AssociateAction::make()
    ->preloadRecordSelect()
```

Scoping the options to associate

You may want to scope the options available to `AssociateAction`:

```
use Filament\Tables\Actions\AssociateAction;
use Illuminate\Database\Eloquent\Builder;

AssociateAction::make()
->recordSelectOptionsQuery(fn (Builder $query) => $query->whereBelongsTo(auth()->user()) )
```

Searching the options to associate across multiple columns

By default, the options available to `AssociateAction` will be searched in the `recordTitleAttribute()` of the table. If you wish to search across multiple columns, you can use the `recordSelectSearchColumns()` method:

```
use Filament\Tables\Actions\AssociateAction;

AssociateAction::make()
->recordSelectSearchColumns(['title', 'description'])
```

Associating multiple records

The `multiple()` method on the `AssociateAction` component allows you to select multiple values:

```
use Filament\Tables\Actions\AssociateAction;

AssociateAction::make()
->multiple()
```

Customizing the select field in the associate modal

You may customize the select field object that is used during association by passing a function to the `recordSelect()` method:

```
use Filament\Forms\Components>Select;
use Filament\Tables\Actions\AssociateAction;

AssociateAction::make()
->recordSelect(
    fn (Select $select) => $select->placeholder('Select a post'),
)
```

Viewing related records

When generating your relation manager, you may pass the `--view` flag to also add a `ViewAction` to the table:

```
php artisan make:filament-relation-manager CategoryResource posts title --view
```

Alternatively, if you've already generated your relation manager, you can just add the `ViewAction` to the `$table-actions()` array:

```
use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->actions([
            Tables\Actions\ViewAction::make(),
            // ...
        ]);
}
```

Deleting related records

By default, you will not be able to interact with deleted records in the relation manager. If you'd like to add functionality to restore, force delete and filter trashed records in your relation manager, use the `--soft-deletes` flag when generating the relation manager:

```
php artisan make:filament-relation-manager CategoryResource posts title --soft-deletes
```

Alternatively, you may add soft deleting functionality to an existing relation manager:

```

use Filament\Tables;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\SoftDeletingScope;

public function table(Table $table): Table
{
    return $table
        ->modifyQueryUsing(fn (Builder $query) => $query->withoutGlobalScopes([
            SoftDeletingScope::class,
        ]))
        ->columns([
            // ...
        ])
        ->filters([
            Tables\Filters\TrashedFilter::make(),
            // ...
        ])
        ->actions([
            Tables\Actions\DeleteAction::make(),
            Tables\Actions\ForceDeleteAction::make(),
            Tables\Actions\RestoreAction::make(),
            // ...
        ])
        ->bulkActions([
            BulkActionGroup::make([
                Tables\Actions\DeleteBulkAction::make(),
                Tables\Actions\ForceDeleteBulkAction::make(),
                Tables\Actions\RestoreBulkAction::make(),
                // ...
            ]),
        ]);
}

```

Customizing the `DeleteAction`

To learn how to customize the `DeleteAction`, including changing the notification and adding lifecycle hooks, please see the [Actions documentation](#).

Importing related records

The `ImportAction` can be added to the header of a relation manager to import records. In this case, you probably want to tell the importer which owner these new records belong to. You can use `import options` to pass through the ID of the owner record:

```

ImportAction::make()
    ->importer(ProductImporter::class)
    ->options(['categoryId' => $this->getOwnerRecord()->getKey()])

```

Now, in the importer class, you can associate the owner in a one-to-many relationship with the imported record:

```
public function resolveRecord(): ?Product
{
    $product = Product::firstOrNew([
        'sku' => $this->data['sku'],
    ]);

    $product->category()->associate($this->options['categoryId']);

    return $product;
}
```

Alternatively, you can attach the record in a many-to-many relationship using the `afterSave()` hook of the importer:

```
protected function afterSave(): void
{
    $this->record->categories()->syncWithoutDetaching([$this->options['categoryId']]);
}
```

Accessing the relationship's owner record

Relation managers are Livewire components. When they are first loaded, the owner record (the Eloquent record which serves as a parent - the main resource model) is saved into a property. You can read this property using:

```
$this->getOwnerRecord()
```

However, if you're inside a `static` method like `form()` or `table()`, `$this` isn't accessible. So, you may use a callback to access the `$livewire` instance:

```
use Filament\Forms;
use Filament\Forms\Form;
use Filament\Resources\RelationManagers\RelationManager;

public function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\Select::make('store_id')
                ->options(function (RelationManager $livewire): array {
                    return $livewire->getOwnerRecord()->stores()
                        ->pluck('name', 'id')
                        ->toArray();
                }),
                // ...
        ]);
}
```

All methods in Filament accept a callback which you can access `$livewire->ownerRecord` in.

Grouping relation managers

You may choose to group relation managers together into one tab. To do this, you may wrap multiple managers in a `RelationGroup` object, with a label:

```
use Filament\Resources\RelationManagers\RelationGroup;

public static function getRelations(): array
{
    return [
        // ...
        RelationGroup::make('Contacts', [
            RelationManagers\IndividualsRelationManager::class,
            RelationManagers\OrganizationsRelationManager::class,
        ]),
        // ...
    ];
}
```

Conditionally showing relation managers

By default, relation managers will be visible if the `viewAny()` method for the related model policy returns `true`.

You may use the `canViewForRecord()` method to determine if the relation manager should be visible for a specific owner record and page:

```
use Illuminate\Database\Eloquent\Model;

public static function canViewForRecord(Model $ownerRecord, string $pageClass): bool
{
    return $ownerRecord->status === Status::Draft;
}
```

Combining the relation manager tabs with the form

On the Edit or View page class, override the `hasCombinedRelationManagerTabsWithContent()` method:

```
public function hasCombinedRelationManagerTabsWithContent(): bool
{
    return true;
}
```

Setting an icon for the form tab

On the Edit or View page class, override the `getContentTabIcon()` method:

```
public function getContentTabIcon(): ?string
{
    return 'heroicon-m-cog';
}
```

Setting the position of the form tab

By default, the form tab is rendered before the relation tabs. To render it after, you can override the `getContentTabPosition()` method on the Edit or View page class:

```
use Filament\Resources\Pages\ContentTabPosition;

public function getContentTabPosition(): ?ContentTabPosition
{
    return ContentTabPosition::After;
}
```

Adding badges to relation manager tabs

You can add a badge to a relation manager tab by setting the `$badge` property:

```
protected static ?string $badge = 'new';
```

Alternatively, you can override the `getBadge()` method:

```
use Illuminate\Database\Eloquent\Model;

public static function getBadge(Model $ownerRecord, string $pageClass): ?string
{
    return static::$badge;
}
```

Or, if you are using a [relation group](#), you can use the `badge()` method:

```
use Filament\Resources\RelationManagers\RelationGroup;

RelationGroup::make('Contacts', [
    // ...
])->badge('new');
```

Changing the color of relation manager tab badges

If a badge value is defined, it will display using the primary color by default. To style the badge contextually, set the `$badgeColor` to either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
protected static ?string $badgeColor = 'danger';
```

Alternatively, you can override the `getBadgeColor()` method:

```
use Illuminate\Database\Eloquent\Model;

public static function getBadgeColor(Model $ownerRecord, string $pageClass): ?string
{
    return 'danger';
}
```

Or, if you are using a [relation group](#), you can use the `badgeColor()` method:

```
use Filament\Resources\RelationManagers\RelationGroup;

RelationGroup::make('Contacts', [
    // ...
])->badgeColor('danger');
```

Adding a tooltip to relation manager tab badges

If a badge value is defined, you can add a tooltip to it by setting the `$badgeTooltip` property:

```
protected static ?string $badgeTooltip = 'There are new posts';
```

Alternatively, you can override the `getBadgeTooltip()` method:

```
use Illuminate\Database\Eloquent\Model;

public static function getBadgeTooltip(Model $ownerRecord, string $pageClass): ?string
{
    return 'There are new posts';
}
```

Or, if you are using a relation group, you can use the `badgeTooltip()` method:

```
use Filament\Resources\RelationManagers\RelationGroup;

RelationGroup::make('Contacts', [
    // ...
])->badgeTooltip('There are new posts');
```

Sharing a resource's form and table with a relation manager

You may decide that you want a resource's form and table to be identical to a relation manager's, and subsequently want to reuse the code you previously wrote. This is easy, by calling the `form()` and `table()` methods of the resource from the relation manager:

```
use App\Filament\Resources\Blog\PostResource;
use Filament\Forms\Form;
use Filament\Tables\Table;

public function form(Form $form): Form
{
    return PostResource::form($form);
}

public function table(Table $table): Table
{
    return PostResource::table($table);
}
```

Hiding a shared form component on the relation manager

If you're sharing a form component from the resource with the relation manager, you may want to hide it on the relation manager. This is especially useful if you want to hide a `Select` field for the owner record in the relation manager, since Filament will handle this for you anyway. To do this, you may use the `hiddenOn()` method, passing the name of the relation manager:

```
use App\Filament\Resources\Blog\PostResource\RelationManagers\CommentsRelationManager;
use Filament\Forms\Components>Select;

Select::make('post_id')
    ->relationship('post', 'title')
    ->hiddenOn(CommentsRelationManager::class)
```

Hiding a shared table column on the relation manager

If you're sharing a table column from the resource with the relation manager, you may want to hide it on the relation manager. This is especially useful if you want to hide a column for the owner record in the relation manager, since this is not appropriate when the owner record is already listed above the relation manager. To do this, you may use the `hiddenOn()` method, passing the name of the relation manager:

```
use App\Filament\Resources\Blog\PostResource\RelationManagers\CommentsRelationManager;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('post.title')
    ->hiddenOn(CommentsRelationManager::class)
```

Hiding a shared table filter on the relation manager

If you're sharing a table filter from the resource with the relation manager, you may want to hide it on the relation manager. This is especially useful if you want to hide a filter for the owner record in the relation manager, since this is not appropriate when the table is already filtered by the owner record. To do this, you may use the `hiddenOn()` method, passing the name of the relation manager:

```
use App\Filament\Resources\Blog\PostResource\RelationManagers\CommentsRelationManager;
use Filament\Tables\Filters>SelectFilter;

SelectFilter::make('post')
    ->relationship('post', 'title')
    ->hiddenOn(CommentsRelationManager::class)
```

Overriding shared configuration on the relation manager

Any configuration that you make inside the resource can be overwritten on the relation manager. For example, if you wanted to disable pagination on the relation manager's inherited table but not the resource itself:

```
use App\Filament\Resources\Blog\PostResource;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return PostResource::table($table)
        ->paginated(false);
}
```

It is probably also useful to provide extra configuration on the relation manager if you wanted to add a header action to [create](#), [attach](#), or [associate](#) records in the relation manager:

```
use App\Filament\Resources\Blog\PostResource;
use Filament\Tables;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return PostResource::table($table)
        ->headerActions([
            Tables\Actions\CreateAction::make(),
            Tables\Actions\AttachAction::make(),
        ]);
}
```

Customizing the relation manager Eloquent query

You can apply your own query constraints or [model scopes](#) that affect the entire relation manager. To do this, you can pass a function to the `modifyQueryUsing()` method of the table, inside which you can customize the query:

```
use Filament\Tables;
use Illuminate\Database\Eloquent\Builder;

public function table(Table $table): Table
{
    return $table
        ->modifyQueryUsing(fn (Builder $query) => $query->where('is_active', true))
        ->columns([
            // ...
        ]);
}
```

Customizing the relation manager title

To set the title of the relation manager, you can use the `$title` property on the relation manager class:

```
protected static ?string $title = 'Posts';
```

To set the title of the relation manager dynamically, you can override the `getTitle()` method on the relation manager class:

```
use Illuminate\Database\Eloquent\Model;

public static function getTitle(Model $ownerRecord, string $pageClass): string
{
    return __('relation-managers.posts.title');
}
```

The title will be reflected in the [heading of the table](#), as well as the relation manager tab if there is more than one. If you want to customize the table heading independently, you can still use the `$table->heading()` method:

```
use Filament\Tables;

public function table(Table $table): Table
{
    return $table
        ->heading('Posts')
        ->columns([
            // ...
        ]);
}
```

Customizing the relation manager record title

The relation manager uses the concept of a "record title attribute" to determine which attribute of the related model should be used to identify it. When creating a relation manager, this attribute is passed as the third argument to the `make:filament-relation-manager` command:

```
php artisan make:filament-relation-manager CategoryResource posts title
```

In this example, the `title` attribute of the `Post` model will be used to identify a post in the relation manager.

This is mainly used by the action classes. For instance, when you attach or associate a record, the titles will be listed in the select field. When you edit, view or delete a record, the title will be used in the header of the modal.

In some cases, you may want to concatenate multiple attributes together to form a title. You can do this by replacing the `recordTitleAttribute()` configuration method with `recordTitle()`, passing a function that transforms a model into a title:

```
use App\Models\Post;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->recordTitle(fn (Post $record): string => "{$record->title} ($record->id}")
        ->columns([
            // ...
        ]);
}
```

If you're using `recordTitle()`, and you have an associate action or attach action, you will also want to specify search columns for those actions:

```
use Filament\Tables\Actions\AssociateAction;
use Filament\Tables\Actions\AttachAction;

AssociateAction::make()
    ->recordSelectSearchColumns(['title', 'id']);

AttachAction::make()
    ->recordSelectSearchColumns(['title', 'id'])
```

Relation pages

Using a `ManageRelatedRecords` page is an alternative to using a relation manager, if you want to keep the functionality of managing a relationship separate from editing or viewing the owner record.

This feature is ideal if you are using [resource sub-navigation](#), as you are easily able to switch between the View or Edit page and the relation page.

To create a relation page, you should use the `make:filament-page` command:

```
php artisan make:filament-page ManageCustomerAddresses --resource=CustomerResource --
type=ManageRelatedRecords
```

When you run this command, you will be asked a series of questions to customize the page, for example, the name of the relationship and its title attribute.

You must register this new page in your resource's `getPages()` method:

```
public static function getPages(): array
{
    return [
        'index' => Pages\ListCustomers::route('/'),
        'create' => Pages\CreateCustomer::route('/create'),
        'view' => Pages\ViewCustomer::route('/{record}'),
        'edit' => Pages>EditCustomer::route('/{record}/edit'),
        'addresses' => Pages\ManageCustomerAddresses::route('/{record}/addresses'),
    ];
}
```

When using a relation page, you do not need to generate a relation manager with `make:filament-relation-manager`, and you do not need to register it in the `getRelations()` method of the resource.

Now, you can customize the page in exactly the same way as a relation manager, with the same `table()` and `form()`.

Adding relation pages to resource sub-navigation

If you're using [resource sub-navigation](#), you can register this page as normal in `getRecordSubNavigation()` of the resource:

```
use App\Filament\Resources\CustomerResource\Pages;
use Filament\Resources\Pages\Page;

public static function getRecordSubNavigation(Page $page): array
{
    return $page->generateNavigationItems([
        // ...
        Pages\ManageCustomerAddresses::class,
    ]);
}
```

Passing properties to relation managers

When registering a relation manager in a resource, you can use the `make()` method to pass an array of [Livewire properties](#) to it:

```
use App\Filament\Resources\Blog\PostResource\RelationManagers\CommentsRelationManager;

public static function getRelations(): array
{
    return [
        CommentsRelationManager::make([
            'status' => 'approved',
        ]),
    ];
}
```

This array of properties gets mapped to public Livewire properties on the relation manager class:

```
use Filament\Resources\RelationManagers\RelationManager;

class CommentsRelationManager extends RelationManager
{
    public string $status;

    // ...
}
```

Now, you can access the `status` in the relation manager class using `$this->status`.

Disabling lazy loading

By default, relation managers are lazy-loaded. This means that they will only be loaded when they are visible on the page.

To disable this behavior, you may override the `$isLazy` property on the relation manager class:

```
protected static bool $isLazy = false;
```

Global Search

Overview

Global search allows you to search across all of your resource records, from anywhere in the app.

Setting global search result titles

To enable global search on your model, you must set a title attribute for your resource:

```
protected static ?string $recordTitleAttribute = 'title';
```

This attribute is used to retrieve the search result title for that record.

Your resource needs to have an Edit or View page to allow the global search results to link to a URL, otherwise no results will be returned for this resource.

You may customize the title further by overriding `getGlobalSearchResultTitle()` method. It may return a plain text string, or an instance of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even Markdown, in the search result title:

```
use Illuminate\Contracts\Support\Htmlable;

public static function getGlobalSearchResultTitle(Model $record): string | Htmlable
{
    return $record->name;
}
```

Globally searching across multiple columns

If you would like to search across multiple columns of your resource, you may override the `getGloballySearchableAttributes()` method. "Dot notation" allows you to search inside relationships:

```
public static function getGloballySearchableAttributes(): array
{
    return ['title', 'slug', 'author.name', 'category.name'];
}
```

Adding extra details to global search results

Search results can display "details" below their title, which gives the user more information about the record. To enable this feature, you must override the `getGlobalSearchResultDetails()` method:

```
public static function getGlobalSearchResultDetails(Model $record): array
{
    return [
        'Author' => $record->author->name,
        'Category' => $record->category->name,
    ];
}
```

In this example, the category and author of the record will be displayed below its title in the search result. However, the `category` and `author` relationships will be lazy-loaded, which will result in poor results performance. To eager-load these relationships, we must override the `getGlobalSearchEloquentQuery()` method:

```
public static function getGlobalSearchEloquentQuery(): Builder
{
    return parent::getGlobalSearchEloquentQuery()->with(['author', 'category']);
}
```

Customizing global search result URLs

Global search results will link to the Edit page of your resource, or the View page if the user does not have edit permissions. To customize this, you may override the `getGlobalSearchResultUrl()` method and return a route of your choice:

```
public static function getGlobalSearchResultUrl(Model $record): string
{
    return UserResource::getUrl('edit', ['record' => $record]);
}
```

Adding actions to global search results

Global search supports actions, which are buttons that render below each search result. They can open a URL or dispatch a Livewire event.

Actions can be defined as follows:

```
use Filament\GlobalSearch\Actions\Action;

public static function getGlobalSearchResultActions(Model $record): array
{
    return [
        Action::make('edit')
            ->url(static::getUrl('edit', ['record' => $record])),
    ];
}
```

You can learn more about how to style action buttons [here](#).

Opening URLs from global search actions

You can open a URL, optionally in a new tab, when clicking on an action:

```
use Filament\GlobalSearch\Actions\Action;

Action::make('view')
    ->url(static::getUrl('view', ['record' => $record]), shouldOpenInNewTab: true)
```

Dispatching Livewire events from global search actions

Sometimes you want to execute additional code when a global search result action is clicked. This can be achieved by setting a Livewire event which should be dispatched on clicking the action. You may optionally pass an array of data, which will be available as parameters in the event listener on your Livewire component:

```
use Filament\GlobalSearch\Actions\Action;

Action::make('quickView')
->dispatch('quickView', [$record->id])
```

Limiting the number of global search results

By default, global search will return up to 50 results per resource. You can customize this on the resource label by overriding the `$globalSearchResultsLimit` property:

```
protected static int $globalSearchResultsLimit = 20;
```

Disabling global search

As [explained above](#), global search is automatically enabled once you set a title attribute for your resource. Sometimes you may want to specify the title attribute while not enabling global search.

This can be achieved by disabling global search in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->globalSearch(false);
}
```

Registering global search key bindings

The global search field can be opened using keyboard shortcuts. To configure these, pass the `globalSearchKeyBindings()` method to the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->globalSearchKeyBindings(['command+k', 'ctrl+k']);
}
```

Configuring the global search debounce

Global search has a default debounce time of 500ms, to limit the number of requests that are made while the user is typing. You can alter this by using the `globalSearchDebounce()` method in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->globalSearchDebounce('750ms');
}
```

Configuring the global search field suffix

Global search field by default doesn't include any suffix. You may customize it using the `globalSearchFieldSuffix()` method in the [configuration](#).

If you want to display the currently configured [global search key bindings](#) in the suffix, you can use the `globalSearchFieldKeyBindingSuffix()` method, which will display the first registered key binding as the suffix of the global search field:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->globalSearchFieldKeyBindingSuffix();
}
```

To customize the suffix yourself, you can pass a string or function to the `globalSearchFieldSuffix()` method. For example, to provide a custom key binding suffix for each platform manually:

```
use Filament\Panel;
use Filament\Support\Enums\Platform;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->globalSearchFieldSuffix(fn (): ?string => match (Platform::detect()) {
            Platform::Windows, Platform::Linux => 'CTRL+K',
            Platform::Mac => '%K',
            default => null,
        });
}
```

Widgets

Overview

Filament allows you to display widgets inside pages, below the header and above the footer.

You can use an existing [dashboard widget](#), or create one specifically for the resource.

Creating a resource widget

To get started building a resource widget:

```
php artisan make:filament-widget CustomerOverview --resource=CustomerResource
```

This command will create two files - a widget class in the `app/Filament/Resources/CustomerResource/Widgets` directory, and a view in the `resources/views/filament/resources/customer-resource/widgets` directory.

You must register the new widget in your resource's `getWidgets()` method:

```
public static function getWidgets(): array
{
    return [
        CustomerResource\Widgets\CustomerOverview::class,
    ];
}
```

If you'd like to learn how to build and customize widgets, check out the [Dashboard](#) documentation section.

Displaying a widget on a resource page

To display a widget on a resource page, use the `getHeaderWidgets()` or `getFooterWidgets()` methods for that page:

```
<?php

namespace App\Filament\Resources\CustomerResource\Pages;

use App\Filament\Resources\CustomerResource;

class ListCustomers extends ListRecords
{
    public static string $resource = CustomerResource::class;

    protected function getHeaderWidgets(): array
    {
        return [
            CustomerResource\Widgets\CustomerOverview::class,
        ];
    }
}
```

`getHeaderWidgets()` returns an array of widgets to display above the page content, whereas `getFooterWidgets()` are displayed below.

If you'd like to customize the number of grid columns used to arrange widgets, check out the [Pages documentation](#).

Accessing the current record in the widget

If you're using a widget on an [Edit](#) or [View](#) page, you may access the current record by defining a `$record` property on the widget class:

```
use Illuminate\Database\Eloquent\Model;

public ?Model $record = null;
```

Accessing page table data in the widget

If you're using a widget on a [List](#) page, you may access the table data by first adding the `ExposesTableToWidgets` trait to the page class:

```
use Filament\Pages\Concerns\ExposesTableToWidgets;
use Filament\Resources\Pages>ListRecords;

class ListProducts extends ListRecords
{
    use ExposesTableToWidgets;

    // ...
}
```

Now, on the widget class, you must add the `InteractsWithPageTable` trait, and return the name of the page class from the `getTablePage()` method:

```
use App\Filament\Resources\ProductResource\Pages>ListProducts;
use Filament\Widgets\Concerns\InteractsWithPageTable;
use Filament\Widgets\Widget;

class ProductStats extends Widget
{
    use InteractsWithPageTable;

    protected function getTablePage(): string
    {
        return ListProducts::class;
    }

    // ...
}
```

In the widget class, you can now access the Eloquent query builder instance for the table data using the `$this->getPageTableQuery()` method:

```
use Filament\Widgets\StatsOverviewWidget\Stat;

Stat::make('Total Products', $this->getPageTableQuery()->count()),
```

Alternatively, you can access a collection of the records on the current page using the `$this->getPageTableRecords()` method:

```
use Filament\Widgets\StatsOverviewWidget\Stat;

Stat::make('Total Products', $this->getPageTableRecords()->count()),
```

Passing properties to widgets on resource pages

When registering a widget on a resource page, you can use the `make()` method to pass an array of [Livewire properties](#) to it:

```
protected function getHeaderWidgets(): array
{
    return [
        CustomerResource\Widgets\CustomerOverview::make([
            'status' => 'active',
        ]),
    ];
}
```

This array of properties gets mapped to [public Livewire properties](#) on the widget class:

```
use Filament\Widgets\Widget;

class CustomerOverview extends Widget
{
    public string $status;

    // ...
}
```

Now, you can access the `status` in the widget class using `$this->status`.

Custom Pages

Overview

Filament allows you to create completely custom pages for resources. To create a new page, you can use:

```
php artisan make:filament-page SortUsers --resource=UserResource --type=custom
```

This command will create two files - a page class in the `/Pages` directory of your resource directory, and a view in the `/pages` directory of the resource views directory.

You must register custom pages to a route in the static `getPages()` method of your resource:

```
public static function getPages(): array
{
    return [
        // ...
        'sort' => Pages\SortUsers::route('/sort'),
    ];
}
```

The order of pages registered in this method matters - any wildcard route segments that are defined before hardcoded ones will be matched by Laravel's router first.

Any parameters defined in the route's path will be available to the page class, in an identical way to [Livewire](#).

Using a resource record

If you'd like to create a page that uses a record similar to the [Edit](#) or [View](#) pages, you can use the `InteractsWithRecord` trait:

```
use Filament\Resources\Pages\Page;
use Filament\Resources\Pages\Concerns\InteractsWithRecord;

class ManageUser extends Page
{
    use InteractsWithRecord;

    public function mount(int | string $record): void
    {
        $this->record = $this->resolveRecord($record);
    }

    // ...
}
```

The `mount()` method should resolve the record from the URL and store it in `$this->record`. You can access the record at any time using `$this->getRecord()` in the class or view.

To add the record to the route as a parameter, you must define `{record}` in `getPages()`:

```
public static function getPages(): array
{
    return [
        // ...
        'manage' => Pages\ManageUser::route('/{record}/manage'),
    ];
}
```

Security

Protecting model attributes

Filament will expose all model attributes to JavaScript, except if they are `$hidden` on your model. This is Livewire's behavior for model binding. We preserve this functionality to facilitate the dynamic addition and removal of form fields after they are initially loaded, while preserving the data they may need.

While attributes may be visible in JavaScript, only those with a form field are actually editable by the user. This is not an issue with mass assignment.

To remove certain attributes from JavaScript on the Edit and View pages, you may override the `mutateFormDataBeforeFill()` method:

```
protected function mutateFormDataBeforeFill(array $data): array
{
    unset($data['is_admin']);

    return $data;
}
```

In this example, we remove the `is_admin` attribute from JavaScript, as it's not being used by the form.

Pages

Overview

Filament allows you to create completely custom pages for the app.

Creating a page

To create a new page, you can use:

```
php artisan make:filament-page Settings
```

This command will create two files - a page class in the `/Pages` directory of the Filament directory, and a view in the `/pages` directory of the Filament views directory.

Page classes are all full-page [Livewire](#) components with a few extra utilities you can use with the panel.

Authorization

You can prevent pages from appearing in the menu by overriding the `canAccess()` method in your Page class. This is useful if you want to control which users can see the page in the navigation, and also which users can visit the page directly:

```
public static function canAccess(): bool
{
    return auth() -> user() -> canManageSettings();
}
```

Adding actions to pages

Actions are buttons that can perform tasks on the page, or visit a URL. You can read more about their capabilities [here](#).

Since all pages are Livewire components, you can [add actions](#) anywhere. Pages already have the `InteractsWithActions` trait, `HasActions` interface, and `<x-filament-actions::modals />` Blade component all set up for you.

Header actions

You can also easily add actions to the header of any page, including [resource pages](#). You don't need to worry about adding anything to the Blade template, we handle that for you. Just return your actions from the `getHeaderActions()` method of the page class:

```
use Filament\Actions\Action;

protected function getHeaderActions(): array
{
    return [
        Action::make('edit')
            ->url(route('posts.edit', ['post' => $this->post])),
        Action::make('delete')
            ->requiresConfirmation()
            ->action(fn () => $this->post->delete()),
    ];
}
```

Opening an action modal when a page loads

You can also open an action when a page loads by setting the `$defaultAction` property to the name of the action you want to open:

```
use Filament\Actions\Action;

public $defaultAction = 'onboarding';

public function onboardingAction(): Action
{
    return Action::make('onboarding')
        ->modalHeading('Welcome')
        ->visible(fn (): bool => ! auth()->user()->isOnBoarded());
}
```

You can also pass an array of arguments to the default action using the `$defaultActionArguments` property:

```
public $defaultActionArguments = ['step' => 2];
```

Alternatively, you can open an action modal when a page loads by specifying the `action` as a query string parameter to the page:

```
/admin/products/edit/932510?action=onboarding
```

Refreshing form data

If you're using actions on an Edit or View resource page, you can refresh data within the main form using the `refreshFormData()` method:

```
use App\Models\Post;
use Filament\Actions\Action;

Action::make('approve')
    ->action(function (Post $record) {
        $record->approve();

        $this->refreshFormData([
            'status',
        ]);
    })
}
```

This method accepts an array of model attributes that you wish to refresh in the form.

Adding widgets to pages

Filament allows you to display [widgets](#) inside pages, below the header and above the footer.

To add a widget to a page, use the `getHeaderWidgets()` or `getFooterWidgets()` methods:

```
use App\Filament\widgets\StatsOverviewWidget;

protected function getHeaderWidgets(): array
{
    return [
        StatsOverviewWidget::class
    ];
}
```

`getHeaderWidgets()` returns an array of widgets to display above the page content, whereas `getFooterWidgets()` are displayed below.

If you'd like to learn how to build and customize widgets, check out the [Dashboard](#) documentation section.

Customizing the widgets' grid

You may change how many grid columns are used to display widgets.

You may override the `getHeaderWidgetsColumns()` or `getFooterWidgetsColumns()` methods to return a number of grid columns to use:

```
public function getHeaderWidgetsColumns(): int | array
{
    return 3;
}
```

Responsive widgets grid

You may wish to change the number of widget grid columns based on the responsive [breakpoint](#) of the browser. You can do this using an array that contains the number of columns that should be used at each breakpoint:

```
public function getHeaderWidgetsColumns(): int | array
{
    return [
        'md' => 4,
        'xl' => 5,
    ];
}
```

This pairs well with [responsive widget widths](#).

Passing data to widgets from the page

You may pass data to widgets from the page using the `getWidgetsData()` method:

```
public function getWidgetData(): array
{
    return [
        'stats' => [
            'total' => 100,
        ],
    ];
}
```

Now, you can define a corresponding public `$stats` array property on the widget class, which will be automatically filled:

```
public $stats = [];
```

Passing properties to widgets on pages

When registering a widget on a page, you can use the `make()` method to pass an array of [Livewire properties](#) to it:

```
use App\Filament\Widgets\StatsOverviewWidget;

protected function getHeaderWidgets(): array
{
    return [
        StatsOverviewWidget::make([
            'status' => 'active',
        ]),
    ];
}
```

This array of properties gets mapped to [public Livewire properties](#) on the widget class:

```
use Filament\Widgets\Widget;

class StatsOverviewWidget extends Widget
{
    public string $status;

    // ...
}
```

Now, you can access the `status` in the widget class using `$this->status`.

Customizing the page title

By default, Filament will automatically generate a title for your page based on its name. You may override this by defining a `$title` property on your page class:

```
protected static ?string $title = 'Custom Page Title';
```

Alternatively, you may return a string from the `getTitle()` method:

```
use Illuminate\Contracts\Support\Htmlable;

public function getTitle(): string | Htmlable
{
    return __('Custom Page Title');
}
```

Customizing the page navigation label

By default, Filament will use the page's `title` as its `navigation` item label. You may override this by defining a `$navigationLabel` property on your page class:

```
protected static ?string $navigationLabel = 'Custom Navigation Label';
```

Alternatively, you may return a string from the `getNavigationLabel()` method:

```
public static function getNavigationLabel(): string
{
    return __('Custom Navigation Label');
}
```

Customizing the page URL

By default, Filament will automatically generate a URL (slug) for your page based on its name. You may override this by defining a `$slug` property on your page class:

```
protected static ?string $slug = 'custom-url-slug';
```

Customizing the page heading

By default, Filament will use the page's `title` as its heading. You may override this by defining a `$heading` property on your page class:

```
protected ?string $heading = 'Custom Page Heading';
```

Alternatively, you may return a string from the `getHeading()` method:

```
public function getHeading(): string
{
    return __('Custom Page Heading');
}
```

Adding a page subheading

You may also add a subheading to your page by defining a `$subheading` property on your page class:

```
protected ?string $subheading = 'Custom Page Subheading';
```

Alternatively, you may return a string from the `getSubheading()` method:

```
public function getSubheading(): ?string
{
    return __('Custom Page Subheading');
}
```

Replacing the page header with a custom view

You may replace the default `heading`, `subheading` and `actions` with a custom header view for any page. You may return it from the `getHeader()` method:

```
use Illuminate\Contracts\View\View;

public function getHeader(): ?View
{
    return view('filament.settings.custom-header');
}
```

This example assumes you have a Blade view at `resources/views/filament/settings/custom-header.blade.php`.

Rendering a custom view in the footer of the page

You may also add a footer to any page, below its content. You may return it from the `getFooter()` method:

```
use Illuminate\Contracts\View\View;

public function getFooter(): ?View
{
    return view('filament.settings.custom-footer');
}
```

This example assumes you have a Blade view at `resources/views/filament/settings/custom-footer.blade.php`.

Customizing the maximum content width

By default, Filament will restrict the width of the content on the page, so it doesn't become too wide on large screens. To change this, you may override the `getMaxContentWidth()` method. Options correspond to [Tailwind's max-width scale](#).

The options are `ExtraSmall`, `Small`, `Medium`, `Large`, `ExtraLarge`, `TwoExtraLarge`, `ThreeExtraLarge`, `FourExtraLarge`, `FiveExtraLarge`, `SixExtraLarge`, `SevenExtraLarge`, `Full`, `MinContent`, `MaxContent`,

`FitContent`, `Prose`, `ScreenSmall`, `ScreenMedium`, `ScreenLarge`, `ScreenExtraLarge` and `ScreenTwoExtraLarge`. The default is `SevenExtraLarge`:

```
use Filament\Support\Enums\MaxWidth;

public function getMaxContentWidth(): MaxWidth
{
    return MaxWidth::Full;
}
```

Generating URLs to pages

Filament provides `getUrl()` static method on page classes to generate URLs to them. Traditionally, you would need to construct the URL by hand or by using Laravel's `route()` helper, but these methods depend on knowledge of the page's slug or route naming conventions.

The `getUrl()` method, without any arguments, will generate a URL:

```
use App\Filament\Pages\Settings;

Settings::getUrl(); // /admin/settings
```

If your page uses URL / query parameters, you should use the argument:

```
use App\Filament\Pages\Settings;

Settings::getUrl(['section' => 'notifications']); // /admin/settings?section=notifications
```

Generating URLs to pages in other panels

If you have multiple panels in your app, `getUrl()` will generate a URL within the current panel. You can also indicate which panel the page is associated with, by passing the panel ID to the `panel` argument:

```
use App\Filament\Pages\Settings;

Settings::getUrl(panel: 'marketing');
```

Adding sub-navigation between pages

You may want to add a common sub-navigation to multiple pages, to allow users to quickly navigate between them. You can do this by defining a cluster. Clusters can also contain resources, and you can switch between multiple pages or resources within a cluster.

Adding extra attributes to the body tag of a page

You may wish to add extra attributes to the `<body>` tag of a page. To do this, you can set an array of attributes in `$extraBodyAttributes`:

```
protected array $extraBodyAttributes = [];
```

Or, you can return an array of attributes and their values from the `getExtraBodyAttributes()` method:

```
public function getExtraBodyAttributes(): array
{
    return [
        'class' => 'settings-page',
    ];
}
```

Dashboard

Overview

Filament allows you to build dynamic dashboards, comprised of "widgets", very easily.

The following document will explain how to use these widgets to assemble a dashboard using the panel.

Available widgets

Filament ships with these widgets:

- [Stats overview](#) widgets display any data, often numeric data, as stats in a row.
- [Chart](#) widgets display numeric data in a visual chart.
- [Table](#) widgets which display a [table](#) on your dashboard.

You may also [create your own custom widgets](#) which can then have a consistent design with Filament's prebuilt widgets.

Sorting widgets

Each widget class contains a `$sort` property that may be used to change its order on the page, relative to other widgets:

```
protected static ?int $sort = 2;
```

Customizing widget width

You may customize the width of a widget using the `$columnSpan` property. You may use a number between 1 and 12 to indicate how many columns the widget should span, or `full` to make it occupy the full width of the page:

```
protected int | string | array $columnSpan = 'full';
```

Responsive widget widths

You may wish to change the widget width based on the responsive [breakpoint](#) of the browser. You can do this using an array that contains the number of columns that the widget should occupy at each breakpoint:

```
protected int | string | array $columnSpan = [
    'md' => 2,
    'xl' => 3,
];
```

This is especially useful when using a [responsive widgets grid](#).

Customizing the widgets' grid

You may change how many grid columns are used to display widgets.

Firstly, you must [replace the original Dashboard page](#).

Now, in your new `app/Filament/Pages/Dashboard.php` file, you may override the `getColumns()` method to return a number of grid columns to use:

```
public function getColumns(): int | string | array
{
    return 2;
}
```

Responsive widgets grid

You may wish to change the number of widget grid columns based on the responsive `breakpoint` of the browser. You can do this using an array that contains the number of columns that should be used at each breakpoint:

```
public function getColumns(): int | string | array
{
    return [
        'md' => 4,
        'xl' => 5,
    ];
}
```

This pairs well with [responsive widget widths](#).

Conditionally hiding widgets

You may override the static `canView()` method on widgets to conditionally hide them:

```
public static function canView(): bool
{
    return auth() -> user() -> isAdmin();
}
```

Table widgets

You may easily add tables to your dashboard. Start by creating a widget with the command:

```
php artisan make:filament-widget LatestOrders --table
```

You may now [customize the table](#) by editing the widget file.

Custom widgets

To get started building a `BlogPostsOverview` widget:

```
php artisan make:filament-widget BlogPostsOverview
```

This command will create two files - a widget class in the `/Widgets` directory of the Filament directory, and a view in the `/widgets` directory of the Filament views directory.

Filtering widget data

You may add a form to the dashboard that allows the user to filter the data displayed across all widgets. When the filters are updated, the widgets will be reloaded with the new data.

Firstly, you must [replace the original Dashboard page](#).

Now, in your new `app/Filament/Pages/Dashboard.php` file, you may add the `HasFiltersForm` trait, and add the `filtersForm()` method to return form components:

```
use Filament\Forms\Components\DatePicker;
use Filament\Forms\Components\Section;
use Filament\Forms\Form;
use Filament\Pages\Dashboard as BaseDashboard;
use Filament\Pages\Dashboard\Concerns\HasFiltersForm;

class Dashboard extends BaseDashboard
{
    use HasFiltersForm;

    public function filtersForm(Form $form): Form
    {
        return $form
            ->schema([
                Section::make()
                    ->schema([
                        DatePicker::make('startDate'),
                        DatePicker::make('endDate'),
                        // ...
                    ])
                    ->columns(3),
            ]);
    }
}
```

In widget classes that require data from the filters, you need to add the `InteractsWithPageFilters` trait, which will allow you to use the `$this->filters` property to access the raw data from the filters form:

```

use App\Models\BlogPost;
use Carbon\CarbonImmutable;
use Filament\Widgets\StatsOverviewWidget;
use Filament\Widgets\Concerns\InteractsWithPageFilters;
use Illuminate\Database\Eloquent\Builder;

class BlogPostsOverview extends StatsOverviewWidget
{
    use InteractsWithPageFilters;

    public function getStats(): array
    {
        $startDate = $this->filters['startDate'] ?? null;
        $endDate = $this->filters['endDate'] ?? null;

        return [
            StatsOverviewWidget\Stat::make(
                label: 'Total posts',
                value: BlogPost::query()
                    ->when($startDate, fn (Builder $query) => $query->whereDate('created_at',
'>=', $startDate))
                    ->when($endDate, fn (Builder $query) => $query->whereDate('created_at',
'<=' , $endDate))
                    ->count(),
            ),
            // ...
        ];
    }
}

```

The `$this->filters` array will always reflect the current form data. Please note that this data is not validated, as it is available live and not intended to be used for anything other than querying the database. You must ensure that the data is valid before using it. In this example, we check if the start date is set before using it in the query.

Filtering widget data using an action modal

Alternatively, you can swap out the filters form for an action modal, that can be opened by clicking a button in the header of the page. There are many benefits to using this approach:

- The filters form is not always visible, which allows you to use the full height of the page for widgets.
- The filters do not update the widgets until the user clicks the "Apply" button, which means that the widgets are not reloaded until the user is ready. This can improve performance if the widgets are expensive to load.
- Validation can be performed on the filters form, which means that the widgets can rely on the fact that the data is valid - the user cannot submit the form until it is. Canceling the modal will discard the user's changes.

To use an action modal instead of a filters form, you can use the `HasFiltersAction` trait instead of `HasFiltersForm`. Then, register the `FilterAction` class as an action in `getHeaderActions()`:

```

use Filament\Forms\Components\DatePicker;
use Filament\Forms\Form;
use Filament\Pages\Dashboard as BaseDashboard;
use Filament\Pages\Dashboard\Actions\FilterAction;
use Filament\Pages\Dashboard\Concerns\HasFiltersAction;

class Dashboard extends BaseDashboard
{
    use HasFiltersAction;

    protected function getHeaderActions(): array
    {
        return [
            FilterAction::make()
                ->form([
                    DatePicker::make('startDate'),
                    DatePicker::make('endDate'),
                    // ...
                ]),
        ];
    }
}

```

Handling data from the filter action is the same as handling data from the filters header form, except that the data is validated before being passed to the widget. The `InteractsWithPageFilters` trait still applies.

Disabling the default widgets

By default, two widgets are displayed on the dashboard. These widgets can be disabled by updating the `widgets()` array of the [configuration](#):

```

use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->widgets([]);
}

```

Customizing the dashboard page

If you want to customize the dashboard class, for example, to [change the number of widget columns](#), create a new file at `app/Filament/Pages/Dashboard.php`:

```

<?php

namespace App\Filament\Pages;

class Dashboard extends \Filament\Pages\Dashboard
{
    // ...
}

```

Finally, remove the original `Dashboard` class from configuration file:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->pages([]);
}
```

Creating multiple dashboards

If you want to create multiple dashboards, you can do so by repeating the process described above. Creating new pages that extend the `Dashboard` class will allow you to create as many dashboards as you need.

You will also need to define the URL path to the extra dashboard, otherwise it will be at `/`:

```
protected static string $routePath = 'finance';
```

You may also customize the title of the dashboard by overriding the `$title` property:

```
protected static ?string $title = 'Finance dashboard';
```

The primary dashboard shown to a user is the first one they have access to (controlled by `canAccess()` method), according to the defined navigation sort order.

The default sort order for dashboards is `-2`. You can control the sort order of custom dashboards with `$navigationSort`:

```
protected static ?int $navigationSort = 15;
```

Navigation

Overview

By default, Filament will register navigation items for each of your [resources](#), [custom pages](#), and [clusters](#). These classes contain static properties and methods that you can override, to configure that navigation item.

If you're looking to add a second layer of navigation to your app, you can use [clusters](#). These are useful for grouping resources and pages together.

Customizing a navigation item's label

By default, the navigation label is generated from the resource or page's name. You may customize this using the `$navigationLabel` property:

```
protected static ?string $navigationLabel = 'Custom Navigation Label';
```

Alternatively, you may override the `getNavigationLabel()` method:

```
public static function getNavigationLabel(): string
{
    return 'Custom Navigation Label';
}
```

Customizing a navigation item's icon

To customize a navigation item's [icon](#), you may override the `$navigationIcon` property on the [resource](#) or [page](#) class:

```
protected static ?string $navigationIcon = 'heroicon-o-document-text';
```

If you set `$navigationIcon = null` on all items within the same navigation group, those items will be joined with a vertical bar below the group label.

Switching navigation item icon when it is active

You may assign a navigation [icon](#) which will only be used for active items using the `$activeNavigationIcon` property:

```
protected static ?string $activeNavigationIcon = 'heroicon-o-document-text';
```

Sorting navigation items

By default, navigation items are sorted alphabetically. You may customize this using the `$navigationSort` property:

```
protected static ?int $navigationSort = 3;
```

Now, navigation items with a lower sort value will appear before those with a higher sort value - the order is ascending.

Adding a badge to a navigation item

To add a badge next to the navigation item, you can use the `getNavigationBadge()` method and return the content of the badge:

```
public static function getNavigationBadge(): ?string
{
    return static::getModel()::count();
}
```

If a badge value is returned by `getNavigationBadge()`, it will display using the primary color by default. To style the badge contextually, return either `danger`, `gray`, `info`, `primary`, `success` or `warning` from the `getNavigationBadgeColor()` method:

```
public static function getNavigationBadgeColor(): ?string
{
    return static::getModel()::count() > 10 ? 'warning' : 'primary';
}
```

A custom tooltip for the navigation badge can be set in `$navigationBadgeTooltip`:

```
protected static ?string $navigationBadgeTooltip = 'The number of users';
```

Or it can be returned from `getNavigationBadgeTooltip()`:

```
public static function getNavigationBadgeTooltip(): ?string
{
    return 'The number of users';
}
```

Grouping navigation items

You may group navigation items by specifying a `$navigationGroup` property on a [resource](#) and [custom page](#):

```
protected static ?string $navigationGroup = 'Settings';
```

All items in the same navigation group will be displayed together under the same group label, "Settings" in this case. Ungrouped items will remain at the start of the navigation.

Grouping navigation items under other items

You may group navigation items as children of other items, by passing the label of the parent item as the `$navigationParentItem`:

```
protected static ?string $navigationParentItem = 'Notifications';

protected static ?string $navigationGroup = 'Settings';
```

You may also use the `getNavigationParentItem()` method to set a dynamic parent item label:

```
public static function getNavigationParentItem(): ?string
{
    return __('filament/navigation.groups.settings.items.notifications');
}
```

As seen above, if the parent item has a navigation group, that navigation group must also be defined, so the correct parent item can be identified.

If you're reaching for a third level of navigation like this, you should consider using [clusters](#) instead, which are a logical grouping of resources and custom pages, which can share their own separate navigation.

Customizing navigation groups

You may customize navigation groups by calling `navigationGroups()` in the [configuration](#), and passing `NavigationGroup` objects in order:

```
use Filament\Navigation\NavigationGroup;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->navigationGroups([
            NavigationGroup::make()
                ->label('Shop')
                ->icon('heroicon-o-shopping-cart'),
            NavigationGroup::make()
                ->label('Blog')
                ->icon('heroicon-o-pencil'),
            NavigationGroup::make()
                ->label(fn (): string => __('navigation.settings'))
                ->icon('heroicon-o-cog-6-tooth')
                ->collapsed(),
        ]);
}
```

In this example, we pass in a custom `icon()` for the groups, and make one `collapsed()` by default.

Ordering navigation groups

By using `navigationGroups()`, you are defining a new order for the navigation groups. If you just want to reorder the groups and not define an entire `NavigationGroup` object, you may just pass the labels of the groups in the new order:

```
$panel
->navigationGroups([
    'Shop',
    'Blog',
    'Settings',
])
```

Making navigation groups not collapsible

By default, navigation groups are collapsible. You may disable this behavior by calling `collapsible(false)` on the `NavigationGroup` object:

```
use Filament\Navigation\NavigationView;

NavigationView::make()
->label('Settings')
->icon('heroicon-o-cog-6-tooth')
->collapsible(false);
```

Or, you can do it globally for all groups in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->collapsibleNavigationGroups(false);
}
```

Adding extra HTML attributes to navigation groups

You can pass extra HTML attributes to the navigation group, which will be merged onto the outer DOM element. Pass an array of attributes to the `extraSidebarAttributes()` or `extraTopbarAttributes()` method, where the key is the attribute name and the value is the attribute value:

```
NavigationView::make()
->extraSidebarAttributes(['class' => 'featured-sidebar-group']),
->extraTopbarAttributes(['class' => 'featured-topbar-group']),
```

The `extraSidebarAttributes()` will be applied to navigation group elements contained in the sidebar, and the `extraTopbarAttributes()` will only be applied to topbar navigation group dropdowns when using [top navigation](#).

Collapsible sidebar on desktop

To make the sidebar collapsible on desktop as well as mobile, you can use the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->sidebarCollapsibleOnDesktop();
}
```

By default, when you collapse the sidebar on desktop, the navigation icons still show. You can fully collapse the sidebar using the `sidebarFullyCollapsibleOnDesktop()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->sidebarFullyCollapsibleOnDesktop();
}
```

Navigation groups in a collapsible sidebar on desktop

This section only applies to `sidebarCollapsibleOnDesktop()`, not `sidebarFullyCollapsibleOnDesktop()`, since the fully collapsible UI just hides the entire sidebar instead of changing its design.

When using a collapsible sidebar on desktop, you will also often be using [navigation groups](#). By default, the labels of each navigation group will be hidden when the sidebar is collapsed, since there is no space to display them. Even if the navigation group itself is [collapsible](#), all items will still be visible in the collapsed sidebar, since there is no group label to click on to expand the group.

These issues can be solved, to achieve a very minimal sidebar design, by [passing an `icon\(\)`](#) to the navigation group objects. When an icon is defined, the icon will be displayed in the collapsed sidebar instead of the items at all times. When the icon is clicked, a dropdown will open to the side of the icon, revealing the items in the group.

When passing an icon to a navigation group, even if the items also have icons, the expanded sidebar UI will not show the item icons. This is to keep the navigation hierarchy clear, and the design minimal. However, the icons for the items will be shown in the collapsed sidebar's dropdowns though, since the hierarchy is already clear from the fact that the dropdown is open.

Registering custom navigation items

To register new navigation items, you can use the [configuration](#):

```
use Filament\Navigation\NavigationItem;
use Filament\Pages\Dashboard;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->navigationItems([
            NavigationItem::make('Analytics')
                ->url('https://filament.pirsch.io', shouldOpenInNewTab: true)
                ->icon('heroicon-o-presentation-chart-line')
                ->group('Reports')
                ->sort(3),
            NavigationItem::make('dashboard')
                ->label(fn (): string => __('filament-panels::pages/dashboard.title'))
                ->url(fn (): string => Dashboard::getUrl())
                ->isActiveWhen(fn () => request()->routeIs('filament.admin.pages.dashboard')) ,
        // ...
    ]);
}
```

Conditionally hiding navigation items

You can also conditionally hide a navigation item by using the `visible()` or `hidden()` methods, passing in a condition to check:

```
use Filament\Navigation\NavigationItem;

NavigationItem::make('Analytics')
->visible(fn(): bool => auth() -> user() -> can('view-analytics'))
// or
->hidden(fn(): bool => ! auth() -> user() -> can('view-analytics')),
```

Disabling resource or page navigation items

To prevent resources or pages from showing up in navigation, you may use:

```
protected static bool $shouldRegisterNavigation = false;
```

Or, you may override the `shouldRegisterNavigation()` method:

```
public static function shouldRegisterNavigation(): bool
{
    return false;
}
```

Please note that these methods do not control direct access to the resource or page. They only control whether the resource or page will show up in the navigation. If you want to also control access, then you should use [resource authorization](#) or [page authorization](#).

Using top navigation

By default, Filament will use a sidebar navigation. You may use a top navigation instead by using the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->topNavigation();
}
```

Customizing the width of the sidebar

You can customize the width of the sidebar by passing it to the `sidebarWidth()` method in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->sidebarWidth('40rem');
}
```

Additionally, if you are using the `sidebarCollapsibleOnDesktop()` method, you can customize width of the collapsed icons by using the `collapsedSidebarWidth()` method in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->sidebarCollapsibleOnDesktop()
        ->collapsedSidebarWidth('9rem');
}
```

Advanced navigation customization

The `navigation()` method can be called from the [configuration](#). It allows you to build a custom navigation that overrides Filament's automatically generated items. This API is designed to give you complete control over the navigation.

Registering custom navigation items

To register navigation items, call the `items()` method:

```

use App\Filament\Pages\Settings;
use App\Filament\Resources\UserResource;
use Filament\Navigation\NavigationBuilder;
use Filament\Navigation\NavigationItem;
use Filament\Pages\Dashboard;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->navigation(function (NavigationBuilder $builder): NavigationBuilder {
            return $builder->items([
                NavigationItem::make('Dashboard')
                    ->icon('heroicon-o-home')
                    ->isActiveWhen(fn (): bool => request()-
>routeIs('filament.admin.pages.dashboard'))
                    ->url(fn (): string => Dashboard::getUrl()),
                    ...UserResource::getNavigationItems(),
                    ...Settings::getNavigationItems(),
            ]);
        });
}

```

Registering custom navigation groups

If you want to register groups, you can call the `groups()` method:

```

use App\Filament\Pages\HomePageSettings;
use App\Filament\Resources\CategoryResource;
use App\Filament\Resources\PageResource;
use Filament\Navigation\NavigationBuilder;
use Filament\Navigation\NavigationGroup;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->navigation(function (NavigationBuilder $builder): NavigationBuilder {
            return $builder->groups([
                NavigationGroup::make('Website')
                    ->items([
                        ...PageResource::getNavigationItems(),
                        ...CategoryResource::getNavigationItems(),
                        ...HomePageSettings::getNavigationItems(),
                    ]),
            ]);
        });
}

```

Disabling navigation

You may disable navigation entirely by passing `false` to the `navigation()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->navigation(false);
}
```

Disabling the navbar

You may disable navbar entirely by passing `false` to the `topbar()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->topbar(false);
}
```

Customizing the user menu

The user menu is featured in the top right corner of the admin layout. It's fully customizable.

To register new items to the user menu, you can use the [configuration](#):

```
use App\Filament\Pages\Settings;
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->userMenuItems([
            MenuItem::make()
                ->label('Settings')
                ->url(fn (): string => Settings::getUrl())
                ->icon('heroicon-o-cog-6-tooth'),
        // ...
    ]);
}
```

Customizing the profile link

To customize the user profile link at the start of the user menu, register a new item with the `profile` array key:

```
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->userMenuItems([
            'profile' => MenuItem::make()->label('Edit profile'),
            // ...
        ]);
}
```

For more information on creating a profile page, check out the [authentication features documentation](#).

Customizing the logout link

To customize the user logout link at the end of the user menu, register a new item with the `logout` array key:

```
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->userMenuItems([
            'logout' => MenuItem::make()->label('Log out'),
            // ...
        ]);
}
```

Conditionally hiding user menu items

You can also conditionally hide a user menu item by using the `visible()` or `hidden()` methods, passing in a condition to check. Passing a function will defer condition evaluation until the menu is actually being rendered:

```
use App\Models\Payment;
use Filament\Navigation\MenuItem;

MenuItem::make()
    ->label('Payments')
    ->visible(fn (): bool => auth()->user()->can('viewAny', Payment::class))
    // or
    ->hidden(fn (): bool => ! auth()->user()->can('viewAny', Payment::class))
```

Sending a `POST` HTTP request from a user menu item

You can send a `POST` HTTP request from a user menu item by passing a URL to the `postAction()` method:

```
use Filament\Navigation\MenuItem;

MenuItem::make()
->label('Lock session')
->postAction(fn (): string => route('lock-session'))
```

Disabling breadcrumbs

The default layout will show breadcrumbs to indicate the location of the current page within the hierarchy of the app.

You may disable breadcrumbs in your [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->breadcrumbs(false);
}
```

Notifications

Overview

The Panel Builder uses the [Notifications](#) package to send messages to users. Please read the [documentation](#) to discover how to send notifications easily.

If you'd like to receive [database notifications](#), you can enable them in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->databaseNotifications();
}
```

You may also control database notification [polling](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->databaseNotifications()
        ->databaseNotificationsPolling('30s');
}
```

Setting up websockets in a panel

The Panel Builder comes with a level of inbuilt support for real-time broadcast and database notifications. However there are a number of areas you will need to install and configure to wire everything up and get it working.

1. If you haven't already, read up on [broadcasting](#) in the Laravel documentation.
2. Install and configure broadcasting to use a [server-side websockets integration](#) like Pusher.
3. If you haven't already, you will need to publish the Filament package configuration:

```
php artisan vendor:publish --tag=filament-config
```

4. Edit the configuration at `config/filament.php` and uncomment the `broadcasting.echo` section - ensuring the settings are correctly configured according to your broadcasting installation.
5. Ensure the [relevant VITE_* entries](#) exist in your `.env` file.
6. Clear relevant caches with `php artisan route:clear` and `php artisan config:clear` to ensure your new configuration takes effect.

Your panel should now be connecting to your broadcasting service. For example, if you log into the Pusher debug console you should see an incoming connection each time you load a page.

To send a real-time notification, see the [broadcast notifications documentation](#).

Users

Overview

By default, all `App\Models\User`s can access Filament locally. To allow them to access Filament in production, you must take a few extra steps to ensure that only the correct users have access to the app.

Authorizing access to the panel

To set up your `App\Models\User` to access Filament in non-local environments, you must implement the `FilamentUser` contract:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Panel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements FilamentUser
{
    // ...

    public function canAccessPanel(Panel $panel): bool
    {
        return str_ends_with($this->email, '@yourdomain.com') && $this->hasVerifiedEmail();
    }
}
```

The `canAccessPanel()` method returns `true` or `false` depending on whether the user is allowed to access the `$panel`. In this example, we check if the user's email ends with `@yourdomain.com` and if they have verified their email address.

Since you have access to the current `$panel`, you can write conditional checks for separate panels. For example, only restricting access to the admin panel while allowing all users to access the other panels of your app:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Panel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements FilamentUser
{
    // ...

    public function canAccessPanel(Panel $panel): bool
    {
        if ($panel->getId() === 'admin') {
            return str_ends_with($this->email, '@yourdomain.com') && $this->hasVerifiedEmail();
        }

        return true;
    }
}
```

Authorizing access to Resources

See the [Authorization](#) section in the Resource documentation for controlling access to Resource pages and their data records.

Setting up user avatars

Out of the box, Filament uses [ui-avatars.com](#) to generate avatars based on a user's name. However, if your user model has an `avatar_url` attribute, that will be used instead. To customize how Filament gets a user's avatar URL, you can implement the `HasAvatar` contract:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Models\Contracts\HasAvatar;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements FilamentUser, HasAvatar
{
    // ...

    public function getFilamentAvatarUrl(): ?string
    {
        return $this->avatar_url;
    }
}
```

The `getFilamentAvatarUrl()` method is used to retrieve the avatar of the current user. If `null` is returned from this method, Filament will fall back to [ui-avatars.com](#).

Using a different avatar provider

You can easily swap out [ui-avatars.com](#) for a different service, by creating a new avatar provider.

In this example, we create a new file at `app/Filament/AvatarProviders/BoringAvatarsProvider.php` for [boringavatars.com](#). The `get()` method accepts a user model instance and returns an avatar URL for that user:

```
<?php

namespace App\Filament\AvatarProviders;

use Filament\AvatarProviders\Contracts;
use Filament\Facades\Filament;
use Illuminate\Contracts\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;

class BoringAvatarsProvider implements Contracts\AvatarProvider
{
    public function get(Model | Authenticatable $record): string
    {
        $name = str(Filament::getNameForDefaultAvatar($record))
            ->trim()
            ->explode(' ')
            ->map(fn (string $segment): string => filled($segment) ? mb_substr($segment, 0, 1) :
        )
            ->join(' ');

        return 'https://source.boringavatars.com/beam/120/' . urlencode($name);
    }
}
```

Now, register this new avatar provider in the [configuration](#):

```
use App\Filament\AvatarProviders\BoringAvatarsProvider;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->defaultAvatarProvider(BoringAvatarsProvider::class);
}
```

Configuring the user's name attribute

By default, Filament will use the `name` attribute of the user to display their name in the app. To change this, you can implement the `HasName` contract:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Models\Contracts\HasName;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements FilamentUser, HasName
{
    // ...

    public function getFilamentName(): string
    {
        return "{$this->first_name} {$this->last_name}";
    }
}
```

The `getFilamentName()` method is used to retrieve the name of the current user.

Authentication features

You can easily enable authentication features for a panel in the configuration file:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->login()
        ->registration()
        ->passwordReset()
        ->emailVerification()
        ->profile();
}
```

Customizing the authentication features

If you'd like to replace these pages with your own, you can pass in any Filament page class to these methods.

Most people will be able to make their desired customizations by extending the base page class from the Filament codebase, overriding methods like `form()`, and then passing the new page class in to the configuration:

```
use App\Filament\Pages\Auth>EditProfile;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->profile(EditProfile::class);
}
```

In this example, we will customize the profile page. We need to create a new PHP class at `app/Filament/Pages/Auth/EditProfile.php`:

```
<?php

namespace App\Filament\Pages\Auth;

use Filament\Forms\Components\TextInput;
use Filament\Forms\Form;
use Filament\Pages\Auth>EditProfile as BaseEditProfile;

class EditProfile extends BaseEditProfile
{
    public function form(Form $form): Form
    {
        return $form
            ->schema([
                TextInput::make('username')
                    ->required()
                    ->maxLength(255),
                $this->getNameFormComponent(),
                $this->getEmailFormComponent(),
                $this->getPasswordFormComponent(),
                $this->getPasswordConfirmationFormComponent(),
            ]);
    }
}
```

This class extends the base profile page class from the Filament codebase. Other page classes you could extend include:

- `Filament\Pages\Auth>Login`
- `Filament\Pages\Auth\Register`
- `Filament\Pages\Auth>EmailVerification\EmailVerificationPrompt`
- `Filament\Pages\Auth>PasswordReset\RequestPasswordReset`
- `Filament\Pages\Auth>PasswordReset\ResetPassword`

In the `form()` method of the example, we call methods like `getNameFormComponent()` to get the default form components for the page. You can customize these components as required. For all the available customization options, see the base `EditProfile` page class in the Filament codebase - it contains all the methods that you can override to make changes.

Customizing an authentication field without needing to re-define the form

If you'd like to customize a field in an authentication form without needing to define a new `form()` method, you could extend the specific field method and chain your customizations:

```
use Filament\Forms\Components\Component;

protected function getPasswordFormComponent(): Component
{
    return parent::getPasswordFormComponent()
        ->reveable(false);
}
```

Using a sidebar on the profile page

By default, the profile page does not use the standard page layout with a sidebar. This is so that it works with the [tenancy](#) feature, otherwise it would not be accessible if the user had no tenants, since the sidebar links are routed to the current tenant.

If you aren't using [tenancy](#) in your panel, and you'd like the profile page to use the standard page layout with a sidebar, you can pass the `isSimple: false` parameter to `$panel->profile()` when registering the page:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->profile(isSimple: false);
}
```

Customizing the authentication route slugs

You can customize the URL slugs used for the authentication routes in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->loginRouteSlug('login')
        ->registrationRouteSlug('register')
        ->passwordResetRoutePrefix('password-reset')
        ->passwordResetRequestRouteSlug('request')
        ->passwordResetRouteSlug('reset')
        ->emailVerificationRoutePrefix('email-verification')
        ->emailVerificationPromptRouteSlug('prompt')
        ->emailVerificationRouteSlug('verify');
}
```

Setting the authentication guard

To set the authentication guard that Filament uses, you can pass in the guard name to the `authGuard()` [configuration](#) method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->authGuard('web');
}
```

Setting the password broker

To set the password broker that Filament uses, you can pass in the broker name to the `authPasswordBroker()` [configuration](#) method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->authPasswordBroker('users');
}
```

Disabling revealable password inputs

By default, all password inputs in authentication forms are `reveable()`. This allows the user can see a plain text version of the password they're typing by clicking a button. To disable this feature, you can pass `false` to the `reveablePasswords()` `configuration` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->reveablePasswords(false);
}
```

You could also disable this feature on a per-field basis by calling `->reveable(false)` on the field object when extending the base page class.

Setting up guest access to a panel

By default, Filament expects to work with authenticated users only. To allow guests to access a panel, you need to avoid using components which expect a signed-in user (such as profiles, avatars), and remove the built-in Authentication middleware:

- Remove the default `Authenticate::class` from the `authMiddleware()` array in the panel configuration.
- Remove `->login()` and any other authentication features from the panel.
- Remove the default `AccountWidget` from the `widgets()` array, because it reads the current user's data.

Authorizing guests in policies

When present, Filament relies on Laravel Model Policies for access control. To give read-access for guest users in a model policy, create the Policy and update the `viewAny()` and `view()` methods, changing the `User $user` param to `?User $user` so that it's optional, and `return true;`. Alternatively, you can remove those methods from the policy entirely.

Configuration

Overview

By default, the configuration file is located at `app/Providers/Filament/AdminPanelProvider.php`. Keep reading to learn more about [panels](#) and how each has [its own configuration file](#).

Introducing panels

By default, when you install the package, there is one panel that has been set up for you - and it lives on `/admin`. All the [resources](#), [custom pages](#), and [dashboard widgets](#) you create get registered to this panel.

However, you can create as many panels as you want, and each can have its own set of resources, pages and widgets.

For example, you could build a panel where users can log in at `/app` and access their dashboard, and admins can log in at `/admin` and manage the app. The `/app` panel and the `/admin` panel have their own resources, since each group of users has different requirements. Filament allows you to do that by providing you with the ability to create multiple panels.

The default admin panel

When you run `filament:install`, a new file is created in `app/Providers/Filament` - `AdminPanelProvider.php`. This file contains the configuration for the `/admin` panel.

When this documentation refers to the "configuration", this is the file you need to edit. It allows you to completely customize the app.

Creating a new panel

To create a new panel, you can use the `make:filament-panel` command, passing in the unique name of the new panel:

```
php artisan make:filament-panel app
```

This command will create a new panel called "app". A configuration file will be created at `app/Providers/Filament/AppPanelProvider.php`. You can access this panel at `/app`, but you can [customize the path](#) if you don't want that.

Since this configuration file is also a [Laravel service provider](#), it needs to be registered in `bootstrap/providers.php` (Laravel 11 and above) or `config/app.php` (Laravel 10 and below). Filament will attempt to do this for you, but if you get an error while trying to access your panel then this process has probably failed.

Changing the path

In a panel configuration file, you can change the path that the app is accessible at using the `path()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->path('app');
}
```

If you want the app to be accessible without any prefix, you can set this to be an empty string:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->path('');
}
```

Make sure your `routes/web.php` file doesn't already define the `''` or `'/'` route, as it will take precedence.

Render hooks

Render hooks allow you to render Blade content at various points in the framework views. You can register global render hooks in a service provider or middleware, but it also allows you to register render hooks that are specific to a panel. To do that, you can use the `renderHook()` method on the panel configuration object. Here's an example, integrating wire-elements/modal with Filament:

```
use Filament\Panel;
use Filament\View\PanelsRenderHook;
use Illuminate\Support\Facades\Blade;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->renderHook(
            PanelsRenderHook::BODY_START,
            fn (): string => Blade::render('@livewire(\'livewire-ui-modal\')'),
        );
}
```

A full list of available render hooks can be found [here](#).

Setting a domain

By default, Filament will respond to requests from all domains. If you'd like to scope it to a specific domain, you can use the `domain()` method, similar to `Route::domain()` in Laravel:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->domain('admin.example.com');
}
```

Customizing the maximum content width

By default, Filament will restrict the width of the content on the page, so it doesn't become too wide on large screens. To change this, you may use the `maxContentWidth()` method. Options correspond to [Tailwind's max-width scale](#). The options are `ExtraSmall`, `Small`, `Medium`, `Large`, `ExtraLarge`, `TwoExtraLarge`, `ThreeExtraLarge`, `FourExtraLarge`, `FiveExtraLarge`, `SixExtraLarge`, `SevenExtraLarge`, `Full`, `MinContent`, `MaxContent`, `FitContent`, `Prose`, `ScreenSmall`, `ScreenMedium`, `ScreenLarge`, `ScreenExtraLarge` and `ScreenTwoExtraLarge`. The default is `SevenExtraLarge`:

```
use Filament\Panel;
use Filament\Support\Enums\MaxWidth;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->maxContentWidth(MaxWidth::Full);
}
```

Lifecycle hooks

Hooks may be used to execute code during a panel's lifecycle. `bootUsing()` is a hook that gets run on every request that takes place within that panel. If you have multiple panels, only the current panel's `bootUsing()` will be run. The function gets run from middleware, after all service providers have been booted:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->bootUsing(function (Panel $panel) {
            // ...
        });
}
```

SPA mode

SPA mode utilizes [Livewire's `wire:navigate`](#) feature to make your server-rendered panel feel like a single-page-application, with less delay between page loads and a loading bar for longer requests. To enable SPA mode on a panel, you can use the `spa()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->spa();
}
```

Disabling SPA navigation for specific URLs

By default, when enabling SPA mode, any URL that lives on the same domain as the current request will be navigated to using Livewire's `wire:navigate` feature. If you want to disable this for specific URLs, you can use the

`spaUrlExceptions()` method:

```
use App\Filament\Resources\PostResource;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->spa()
        ->spaUrlExceptions(fn (): array => [
            url('/admin'),
            PostResource::getUrl(),
        ]);
}
```

In this example, we are using `getUrl()` on a resource to get the URL to the resource's index page. This feature requires the panel to already be registered though, and the configuration is too early in the request lifecycle to do that. You can use a function to return the URLs instead, which will be resolved when the panel has been registered.

These URLs need to exactly match the URL that the user is navigating to, including the domain and protocol. If you'd like to use a pattern to match multiple URLs, you can use an asterisk (`*`) as a wildcard character:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->spa()
        ->spaUrlExceptions([
            '*/admin/posts/*',
        ]);
}
```

Unsaved changes alerts

You may alert users if they attempt to navigate away from a page without saving their changes. This is applied on [Create](#) and [Edit](#) pages of a resource, as well as any open action modals. To enable this feature, you can use the

`unsavedChangesAlerts()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->unsavedChangesAlerts();
}
```

Enabling database transactions

By default, Filament does not wrap operations in database transactions, and allows the user to enable this themselves when they have tested to ensure that their operations are safe to be wrapped in a transaction. However, you can enable

database transactions at once for all operations by using the `databaseTransactions()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->databaseTransactions();
}
```

For any actions you do not want to be wrapped in a transaction, you can use the `databaseTransaction(false)` method:

```
CreateAction::make()
    ->databaseTransaction(false)
```

And for any pages like [Create resource](#) and [Edit resource](#), you can define the `$hasDatabaseTransactions` property to `false` on the page class:

```
use Filament\Resources\Pages\CreateRecord;

class CreatePost extends CreateRecord
{
    protected ?bool $hasDatabaseTransactions = false;

    // ...
}
```

Opting in to database transactions for specific actions and pages

Instead of enabling database transactions everywhere and opting out of them for specific actions and pages, you can opt in to database transactions for specific actions and pages.

For actions, you can use the `databaseTransaction()` method:

```
CreateAction::make()
    ->databaseTransaction()
```

For pages like [Create resource](#) and [Edit resource](#), you can define the `$hasDatabaseTransactions` property to `true` on the page class:

```
use Filament\Resources\Pages\CreateRecord;

class CreatePost extends CreateRecord
{
    protected ?bool $hasDatabaseTransactions = true;

    // ...
}
```

Registering assets for a panel

You can register `assets` that will only be loaded on pages within a specific panel, and not in the rest of the app. To do that, pass an array of assets to the `assets()` method:

```
use Filament\Panel;
use Filament\Support\Assets\Css;
use Filament\Support\Assets\Js;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->assets([
            Css::make('custom-stylesheet', resource_path('css/custom.css')),
            Js::make('custom-script', resource_path('js/custom.js')),
        ]);
}
```

Applying middleware

You can apply extra middleware to all routes by passing an array of middleware classes to the `middleware()` method in the configuration:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->middleware([
            // ...
        ]);
}
```

By default, middleware will be run when the page is first loaded, but not on subsequent Livewire AJAX requests. If you want to run middleware on every request, you can make it persistent by passing `true` as the second argument to the `middleware()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->middleware([
            // ...
        ], isPersistent: true);
}
```

Applying middleware to authenticated routes

You can apply middleware to all authenticated routes by passing an array of middleware classes to the `authMiddleware()` method in the configuration:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->authMiddleware([
            // ...
        ]);
}
```

By default, middleware will be run when the page is first loaded, but not on subsequent Livewire AJAX requests. If you want to run middleware on every request, you can make it persistent by passing `true` as the second argument to the `authMiddleware()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->authMiddleware([
            // ...
        ], isPersistent: true);
}
```

Disabling broadcasting

By default, Laravel Echo will automatically connect for every panel, if credentials have been set up in the [published config/filament.php configuration file](#). To disable this automatic connection in a panel, you can use the `broadcasting(false)` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->broadcasting(false);
}
```

Clusters

Overview

Clusters are a hierarchical structure in panels that allow you to group [resources](#) and [custom pages](#) together. They are useful for organizing your panel into logical sections, and can help reduce the size of your panel's sidebar.

When using a cluster, a few things happen:

- A new navigation item is added to the navigation, which is a link to the first resource or page in the cluster.
- The individual navigation items for the resources or pages are no longer visible in the main navigation.
- A new sub-navigation UI is added to each resource or page in the cluster, which contains the navigation items for the resources or pages in the cluster.
- Resources and pages in the cluster get a new URL, prefixed with the name of the cluster. If you are generating URLs to [resources](#) and [pages](#) correctly, then this change should be handled for you automatically.
- The cluster's name is in the breadcrumbs of all resources and pages in the cluster. When clicking it, you are taken to the first resource or page in the cluster.

Creating a cluster

Before creating your first cluster, you must tell the panel where cluster classes should be located. Alongside methods like `discoverResources()` and `discoverPages()` in the [configuration](#), you can use `discoverClusters()`:

```
public function panel(Panel $panel): Panel
{
    return $panel
    // ...
    ->discoverResources(in: app_path('Filament/Resources'), for: 'App\Filament\Resources')
    ->discoverPages(in: app_path('Filament/Pages'), for: 'App\Filament\Pages')
    ->discoverClusters(in: app_path('Filament/Clusters'), for: 'App\Filament\Clusters');
}
```

Now, you can create a cluster with the `php artisan make:filament-cluster` command:

```
php artisan make:filament-cluster Settings
```

This will create a new cluster class in the `app/Filament/Clusters` directory:

```
<?php

namespace App\Filament\Clusters;

use Filament\Clusters\Cluster;

class Settings extends Cluster
{
    protected static ?string $navigationIcon = 'heroicon-o-squares-2x2';
}
```

The `$navigationIcon` property is defined by default since you will most likely want to customize this immediately. All other [navigation properties and methods](#) are also available to use, including `$navigationLabel`, `$navigationSort`

and `$navigationGroup`. These are used to customize the cluster's main navigation item, in the same way you would customize the item for a resource or page.

Adding resources and pages to a cluster

To add resources and pages to a cluster, you just need to define the `$cluster` property on the resource or page class, and set it to the cluster class [you created](#):

```
use App\Filament\Clusters\Settings;

protected static ?string $cluster = Settings::class;
```

Code structure recommendations for panels using clusters

When using clusters, it is recommended that you move all of your resources and pages into a directory with the same name as the cluster. For example, here is a directory structure for a panel that uses a cluster called `Settings`, containing a `ColorResource` and two custom pages:

```
.
+-- Clusters
|   +-- Settings.php
|   +-- Settings
|   |   +-- Pages
|   |   |   +-- ManageBranding.php
|   |   |   +-- ManageNotifications.php
|   |   +-- Resources
|   |   |   +-- ColorResource.php
|   |   |   +-- ColorResource
|   |   |   |   +-- Pages
|   |   |   |   |   +-- CreateColor.php
|   |   |   |   |   +-- EditColor.php
|   |   |   |   |   +-- ListColors.php
```

This is a recommendation, not a requirement. You can structure your panel however you like, as long as the resources and pages in your cluster use the `$cluster` property. This is just a suggestion to help you keep your panel organized.

When a cluster exists in your panel, and you generate new resources or pages with the `make:filament-resource` or `make:filament-page` commands, you will be asked if you want to create them inside a cluster directory, according to these guidelines. If you choose to, then Filament will also assign the correct `$cluster` property to the resource or page class for you. If you do not, you will need to [define the `\$cluster` property](#) yourself.

Customizing the cluster breadcrumb

The cluster's name is in the breadcrumbs of all resources and pages in the cluster.

You may customize the breadcrumb name using the `$clusterBreadcrumb` property in the cluster class:

```
protected static ?string $clusterBreadcrumb = 'cluster';
```

Alternatively, you may use the `getClusterBreadcrumb()` to define a dynamic breadcrumb name:

```
public static function getClusterBreadcrumb(): string
{
    return ___('filament/clusters/cluster.name');
}
```

Tenancy

Overview

Multi-tenancy is a concept where a single instance of an application serves multiple customers. Each customer has their own data and access rules that prevent them from viewing or modifying each other's data. This is a common pattern in SaaS applications. Users often belong to groups of users (often called teams or organizations). Records are owned by the group, and users can be members of multiple groups. This is suitable for applications where users need to collaborate on data.

Multi-tenancy is a very sensitive topic. It's important to understand the security implications of multi-tenancy and how to properly implement it. If implemented partially or incorrectly, data belonging to one tenant may be exposed to another tenant. Filament provides a set of tools to help you implement multi-tenancy in your application, but it is up to you to understand how to use them. Filament does not provide any guarantees about the security of your application. It is your responsibility to ensure that your application is secure. Please see the [security](#) section for more information.

Simple one-to-many tenancy

The term "multi-tenancy" is broad and may mean different things in different contexts. Filament's tenancy system implies that the user belongs to **many** tenants (*organizations, teams, companies, etc.*) and may switch between them.

If your case is simpler and you don't need a many-to-many relationship, then you don't need to set up the tenancy in Filament. You could use [observers](#) and [global scopes](#) instead.

Let's say you have a database column `[users.team_id]`, you can scope all records to have the same `[team_id]` as the user using a [global scope](#):

```
use Illuminate\Database\Eloquent\Builder;

class Post extends Model
{
    protected static function booted(): void
    {
        static::addGlobalScope('team', function (Builder $query) {
            if (auth()->hasUser()) {
                $query->where('team_id', auth()->user()->team_id);
                // or with a `team` relationship defined:
                $query->whereBelongsTo(auth()->user()->team);
            }
        });
    }
}
```

To automatically set the `[team_id]` on the record when it's created, you can create an [observer](#):

```
class PostObserver
{
    public function creating(Post $post): void
    {
        if ($auth()->hasUser()) {
            $post->team_id = $auth()->user()->team_id;
            // or with a `team` relationship defined:
            $post->team() ->associate($auth()->user()->team);
        }
    }
}
```

Setting up tenancy

To set up tenancy, you'll need to specify the "tenant" (like team or organization) model in the [configuration](#):

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenant(Team::class);
}
```

You'll also need to tell Filament which tenants a user belongs to. You can do this by implementing the [HasTenants](#) interface on the [App\Models\User](#) model:

```

<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Models\Contracts\HasTenants;
use Filament\Panel;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Collection;

class User extends Authenticatable implements FilamentUser, HasTenants
{
    // ...

    public function teams(): BelongsToMany
    {
        return $this->belongsToMany(Team::class);
    }

    public function getTenants(Panel $panel): Collection
    {
        return $this->teams();
    }

    public function canAccessTenant(Model $tenant): bool
    {
        return $this->teams()->whereKey($tenant)->exists();
    }
}

```

In this example, users belong to many teams, so there is a `teams()` relationship. The `getTenants()` method returns the teams that the user belongs to. Filament uses this to list the tenants that the user has access to.

For security, you also need to implement the `canAccessTenant()` method of the `HasTenants` interface to prevent users from accessing the data of other tenants by guessing their tenant ID and putting it into the URL.

You'll also want users to be able to [register new teams](#).

Adding a tenant registration page

A registration page will allow users to create a new tenant.

When visiting your app after logging in, users will be redirected to this page if they don't already have a tenant.

To set up a registration page, you'll need to create a new page class that extends `Filament\Pages\Tenancy\RegisterTenant`. This is a full-page Livewire component. You can put this anywhere you want, such as `app/Filament/Pages/Tenancy/RegisterTeam.php`:

```

namespace App\Filament\Pages\Tenancy;

use App\Models\Team;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Form;
use Filament\Pages\Tenancy\RegisterTenant;

class RegisterTeam extends RegisterTenant
{
    public static function getLabel(): string
    {
        return 'Register team';
    }

    public function form(Form $form): Form
    {
        return $form
            ->schema([
                TextInput::make('name'),
                // ...
            ]);
    }

    protected function handleRegistration(array $data): Team
    {
        $team = Team::create($data);

        $team->members()->attach(auth()->user());

        return $team;
    }
}

```

You may add any [form components](#) to the `form()` method, and create the team inside the `handleRegistration()` method.

Now, we need to tell Filament to use this page. We can do this in the [configuration](#):

```

use App\Filament\Pages\Tenancy\RegisterTeam;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantRegistration(RegisterTeam::class);
}

```

Customizing the tenant registration page

You can override any method you want on the base registration page class to make it act as you want. Even the `$view` property can be overridden to use a custom view of your choice.

Adding a tenant profile page

A profile page will allow users to edit information about the tenant.

To set up a profile page, you'll need to create a new page class that extends `Filament\Pages\Tenancy>EditTenantProfile`. This is a full-page Livewire component. You can put this anywhere you want, such as `app/Filament/Pages/Tenancy/EditTeamProfile.php`:

```
namespace App\Filament\Pages\Tenancy;

use Filament\Forms\Components\TextInput;
use Filament\Forms\Form;
use Filament\Pages\Tenancy>EditTenantProfile;

class EditTeamProfile extends EditTenantProfile
{
    public static function getLabel(): string
    {
        return 'Team profile';
    }

    public function form(Form $form): Form
    {
        return $form
            ->schema([
                TextInput::make('name'),
                // ...
            ]);
    }
}
```

You may add any form components to the `form()` method. They will get saved directly to the tenant model.

Now, we need to tell Filament to use this page. We can do this in the configuration:

```
use App\Filament\Pages\Tenancy>EditTeamProfile;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantProfile(EditTeamProfile::class);
}
```

Customizing the tenant profile page

You can override any method you want on the base profile page class to make it act as you want. Even the `$view` property can be overridden to use a custom view of your choice.

Accessing the current tenant

Anywhere in the app, you can access the tenant model for the current request using `Filament::getTenant()`:

```
use Filament\Facades\Filament;

$tenant = Filament::getTenant();
```

Billing

Using Laravel Spark

Filament provides a billing integration with [Laravel Spark](#). Your users can start subscriptions and manage their billing information.

To install the integration, first [install Spark](#) and configure it for your tenant model.

Now, you can install the Filament billing provider for Spark using Composer:

```
composer require filament/spark-billing-provider
```

In the [configuration](#), set Spark as the `tenantBillingProvider()`:

```
use Filament\Billing\Providers\SparkBillingProvider;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantBillingProvider(new SparkBillingProvider());
}
```

Now, you're all good to go! Users can manage their billing by clicking a link in the tenant menu.

Requiring a subscription

To require a subscription to use any part of the app, you can use the `requiresTenantSubscription()` configuration method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->requiresTenantSubscription();
}
```

Now, users will be redirected to the billing page if they don't have an active subscription.

Requiring a subscription for specific resources and pages

Sometimes, you may wish to only require a subscription for certain [resources](#) and [custom pages](#) in your app. You can do this by returning `true` from the `isTenantSubscriptionRequired()` method on the resource or page class:

```
public static function isTenantSubscriptionRequired(Panel $panel): bool
{
    return true;
}
```

If you're using the `requiresTenantSubscription()` configuration method, then you can return `false` from this method to allow access to the resource or page as an exception.

Writing a custom billing integration

Billing integrations are quite simple to write. You just need a class that implements the `Filament\Billing\Contracts\Provider` interface. This interface has two methods.

`getRouteAction()` is used to get the route action that should be run when the user visits the billing page. This could be a callback function, or the name of a controller, or a Livewire component - anything that works when using `Route::get()` in Laravel normally. For example, you could put in a simple redirect to your own billing page using a callback function.

`getSubscribedMiddleware()` returns the name of a middleware that should be used to check if the tenant has an active subscription. This middleware should redirect the user to the billing page if they don't have an active subscription.

Here's an example billing provider that uses a callback function for the route action and a middleware for the subscribed middleware:

```
use App\Http\Middleware\RedirectIfUserNotSubscribed;
use Filament\Billing\Contracts\Provider;
use Illuminate\Http\RedirectResponse;

class ExampleBillingProvider implements Provider
{
    public function getRouteAction(): string
    {
        return function (): RedirectResponse {
            return redirect('https://billing.example.com');
        };
    }

    public function getSubscribedMiddleware(): string
    {
        return RedirectIfUserNotSubscribed::class;
    }
}
```

Customizing the billing route slug

You can customize the URL slug used for the billing route using the `tenantBillingRouteSlug()` method in the [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantBillingRouteSlug('billing');
}
```

Customizing the tenant menu

The tenant-switching menu is featured in the admin layout. It's fully customizable.

To register new items to the tenant menu, you can use the [configuration](#):

```
use App\Filament\Pages\Settings;
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMenuItems([
            MenuItem::make()
                ->label('Settings')
                ->url(fn (): string => Settings::getUrl())
                ->icon('heroicon-m-cog-8-tooth'),
        // ...
    ]);
}
```

Customizing the registration link

To customize the registration link on the tenant menu, register a new item with the `register` array key:

```
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMenuItems([
            'register' => MenuItem::make()->label('Register new team'),
        // ...
    ]);
}
```

Customizing the profile link

To customize the profile link on the tenant menu, register a new item with the `profile` array key:

```
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMenuItems([
            'profile' => MenuItem::make()->label('Edit team profile'),
            // ...
        ]);
}
```

Customizing the billing link

To customize the billing link on the tenant menu, register a new item with the `billing` array key:

```
use Filament\Navigation\MenuItem;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMenuItems([
            'billing' => MenuItem::make()->label('Manage subscription'),
            // ...
        ]);
}
```

Conditionally hiding tenant menu items

You can also conditionally hide a tenant menu item by using the `visible()` or `hidden()` methods, passing in a condition to check. Passing a function will defer condition evaluation until the menu is actually being rendered:

```
use Filament\Navigation\MenuItem;

MenuItem::make()
    ->label('Settings')
    ->visible(fn (): bool => auth()->user()->can('manage-team'))
    // or
    ->hidden(fn (): bool => ! auth()->user()->can('manage-team'))
```

Sending a `POST` HTTP request from a tenant menu item

You can send a `POST` HTTP request from a tenant menu item by passing a URL to the `postAction()` method:

```
use Filament\Navigation\MenuItem;

MenuItem::make()
    ->label('Lock session')
    ->postAction(fn (): string => route('lock-session'))
```

Hiding the tenant menu

You can hide the tenant menu by using the `tenantMenu(false)`

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMenu(false);
}
```

However, this is a sign that Filament's tenancy feature is not suitable for your project. If each user only belongs to one tenant, you should stick to [simple one-to-many tenancy](#).

Setting up avatars

Out of the box, Filament uses [ui-avatars.com](#) to generate avatars based on a user's name. However, if your user model has an `avatar_url` attribute, that will be used instead. To customize how Filament gets a user's avatar URL, you can implement the `HasAvatar` contract:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Models\Contracts\HasAvatar;
use Illuminate\Database\Eloquent\Model;

class Team extends Model implements HasAvatar
{
    // ...

    public function getFilamentAvatarUrl(): ?string
    {
        return $this->avatar_url;
    }
}
```

The `getFilamentAvatarUrl()` method is used to retrieve the avatar of the current user. If `null` is returned from this method, Filament will fall back to [ui-avatars.com](#).

You can easily swap out [ui-avatars.com](#) for a different service, by creating a new avatar provider. [You can learn how to do this here.](#)

Configuring the tenant relationships

When creating and listing records associated with a Tenant, Filament needs access to two Eloquent relationships for each resource - an "ownership" relationship that is defined on the resource model class, and a relationship on the tenant model class. By default, Filament will attempt to guess the names of these relationships based on standard Laravel conventions. For example, if the tenant model is `App\Models\Team`, it will look for a `team()` relationship on the resource model class. And if the resource model class is `App\Models\Post`, it will look for a `posts()` relationship on the tenant model class.

Customizing the ownership relationship name

You can customize the name of the ownership relationship used across all resources at once, using the `$ownershipRelationship` argument on the `tenant()` configuration method. In this example, resource model classes have an `owner` relationship defined:

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenant(Team::class, ownershipRelationship: 'owner');
}
```

Alternatively, you can set the `$tenantOwnershipRelationshipName` static property on the resource class, which can then be used to customize the ownership relationship name that is just used for that resource. In this example, the `Post` model class has an `owner` relationship defined:

```
use Filament\Resources\Resource;

class PostResource extends Resource
{
    protected static ?string $tenantOwnershipRelationshipName = 'owner';

    // ...
}
```

Customizing the resource relationship name

You can set the `$tenantRelationshipName` static property on the resource class, which can then be used to customize the relationship name that is used to fetch that resource. In this example, the tenant model class has an `blogPosts` relationship defined:

```
use Filament\Resources\Resource;

class PostResource extends Resource
{
    protected static ?string $tenantRelationshipName = 'blogPosts';

    // ...
}
```

Configuring the slug attribute

When using a tenant like a team, you might want to add a slug field to the URL rather than the team's ID. You can do that with the `slugAttribute` argument on the `tenant()` configuration method:

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenant(Team::class, slugAttribute: 'slug');
}
```

Configuring the name attribute

By default, Filament will use the `name` attribute of the tenant to display its name in the app. To change this, you can implement the `HasName` contract:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\HasName;
use Illuminate\Database\Eloquent\Model;

class Team extends Model implements HasName
{
    // ...

    public function getFilamentName(): string
    {
        return "{$this->name} {$this->subscription_plan}";
    }
}
```

The `getFilamentName()` method is used to retrieve the name of the current user.

Setting the current tenant label

Inside the tenant switcher, you may wish to add a small label like "Active team" above the name of the current team. You can do this by implementing the `HasCurrentTenantLabel` method on the tenant model:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\HasCurrentTenantLabel;
use Illuminate\Database\Eloquent\Model;

class Team extends Model implements HasCurrentTenantLabel
{
    // ...

    public function getCurrentTenantLabel(): string
    {
        return 'Active team';
    }
}
```

Setting the default tenant

When signing in, Filament will redirect the user to the first tenant returned from the `getTenants()` method.

Sometimes, you might wish to change this. For example, you might store which team was last active, and redirect the user to that team instead.

To customize this, you can implement the `HasDefaultTenant` contract on the user:

```
<?php

namespace App\Models;

use Filament\Models\Contracts\FilamentUser;
use Filament\Models\Contracts\HasDefaultTenant;
use Filament\Models\Contracts\HasTenants;
use Filament\Panel;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class User extends Model implements FilamentUser, HasDefaultTenant, HasTenants
{
    // ...

    public function getDefaultTenant(Panel $panel): ?Model
    {
        return $this->latestTeam();
    }

    public function latestTeam(): BelongsTo
    {
        return $this->belongsTo(Team::class, 'latest_team_id');
    }
}
```

Applying middleware to tenant-aware routes

You can apply extra middleware to all tenant-aware routes by passing an array of middleware classes to the `tenantMiddleware()` method in the [panel configuration file](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMiddleware([
            // ...
        ]);
}
```

By default, middleware will be run when the page is first loaded, but not on subsequent Livewire AJAX requests. If you want to run middleware on every request, you can make it persistent by passing `true` as the second argument to the `tenantMiddleware()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMiddleware([
            // ...
        ], isPersistent: true);
}
```

Adding a tenant route prefix

By default the URL structure will put the tenant ID or slug immediately after the panel path. If you wish to prefix it with another URL segment, use the `tenantRoutePrefix()` method:

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->path('admin')
        ->tenant(Team::class)
        ->tenantRoutePrefix('team');
}
```

Before, the URL structure was `/admin/1` for tenant 1. Now, it is `/admin/team/1`.

Using a domain to identify the tenant

When using a tenant, you might want to use domain or subdomain routing like `team1.example.com/posts` instead of a route prefix like `/team1/posts`. You can do that with the `tenantDomain()` method, alongside the `tenant()` configuration method. The `tenant` argument corresponds to the slug attribute of the tenant model:

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenant(Team::class, slugAttribute: 'slug')
        ->tenantDomain('{tenant:slug}.example.com');
}
```

In the above examples, the tenants live on subdomains of the main app domain. You may also set the system up to resolve the entire domain from the tenant as well:

```
use App\Models\Team;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenant(Team::class, slugAttribute: 'domain')
        ->tenantDomain('{tenant:domain}');
}
```

In this example, the `domain` attribute should contain a valid domain host, like `example.com` or `subdomain.example.com`.

Note: When using a parameter for the entire domain (`{tenantDomain('{tenant:domain}')`), Filament will register a global route parameter pattern for all `[tenant]` parameters in the application to be `[a-z0-9.\-]+`. This is because Laravel does not allow the `.` character in route parameters by default. This might conflict with other panels using tenancy, or other parts of your application that use a `[tenant]` route parameter.

Disabling tenancy for a resource

By default, all resources within a panel with tenancy will be scoped to the current tenant. If you have resources that are shared between tenants, you can disable tenancy for them by setting the `$isScopedToTenant` static property to `false` on the resource class:

```
protected static bool $isScopedToTenant = false;
```

Disabling tenancy for all resources

If you wish to opt-in to tenancy for each resource instead of opting-out, you can call `Resource::scopeToTenant(false)` inside a service provider's `boot()` method or a middleware:

```
use Filament\Resources\Resource;

Resource::scopeToTenant(false);
```

Now, you can opt-in to tenancy for each resource by setting the `$isScopedToTenant` static property to `true` on a resource class:

```
protected static bool $isScopedToTenant = true;
```

Tenancy security

It's important to understand the security implications of multi-tenancy and how to properly implement it. If implemented partially or incorrectly, data belonging to one tenant may be exposed to another tenant. Filament provides a set of tools to help you implement multi-tenancy in your application, but it is up to you to understand how to use them. Filament does not provide any guarantees about the security of your application. It is your responsibility to ensure that your application is secure.

Below is a list of features that Filament provides to help you implement multi-tenancy in your application:

- Automatic scoping of resources to the current tenant. The base Eloquent query that is used to fetch records for a resource is automatically scoped to the current tenant. This query is used to render the resource's list table, and is also used to resolve records from the current URL when editing or viewing a record. This means that if a user attempts to view a record that does not belong to the current tenant, they will receive a 404 error.
- Automatic association of new resource records to the current tenant.

And here are the things that Filament does not currently provide:

- Scoping of relation manager records to the current tenant. When using the relation manager, in the vast majority of cases, the query will not need to be scoped to the current tenant, since it is already scoped to the parent record, which is itself scoped to the current tenant. For example, if a `Team` tenant model had an `Author` resource, and that resource had a `posts` relationship and relation manager set up, and posts only belong to one author, there is no need to scope the query. This is because the user will only be able to see authors that belong to the current team anyway, and thus will only be able to see posts that belong to those authors. You can [scope the Eloquent query](#) if you wish.
- Form component and filter scoping. When using the `Select`, `CheckboxList` or `Repeater` form components, the `SelectFilter`, or any other similar Filament component which is able to automatically fetch "options" or other data from the database (usually using a `relationship()` method), this data is not scoped. The main reason for this is that these features often don't belong to the Filament Panel Builder package, and have no knowledge that they are being used within that context, and that a tenant even exists. And even if they did have access to the tenant, there is nowhere for the tenant relationship configuration to live. To scope these components, you need to pass in a query function that scopes the query to the current tenant. For example, if you were using the `Select` form component to select an `author` from a relationship, you could do this:

```
use Filament\Facades\Filament;
use Filament\Forms\Components\Select;
use Illuminate\Database\Eloquent\Builder;

Select::make('author_id')
    ->relationship(
        name: 'author',
        titleAttribute: 'name',
        modifyQueryUsing: fn (Builder $query) => $query->whereBelongsTo(Filament::getTenant()),
    );
```

Using tenant-aware middleware to apply global scopes

It might be useful to apply global scopes to your Eloquent models while they are being used in your panel. This would allow you to forget about scoping your queries to the current tenant, and instead have the scoping applied automatically. To do this, you can create a new middleware class like `ApplyTenantScopes`:

```
php artisan make:middleware ApplyTenantScopes
```

Inside the `handle()` method, you can apply any global scopes that you wish:

```
use App\Models\Author;
use Closure;
use Filament\Facades\Filament;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Http\Request;

class ApplyTenantScopes
{
    public function handle(Request $request, Closure $next)
    {
        Author::addGlobalScope(
            fn (Builder $query) => $query->whereBelongsTo(Filament::getTenant()),
        );
    }

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

You can now [register this middleware](#) for all tenant-aware routes, and ensure that it is used across all Livewire AJAX requests by making it persistent:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->tenantMiddleware([
            ApplyTenantScopes::class,
        ], isPersistent: true);
}
```

Themes

Changing the colors

In the [configuration](#), you can easily change the colors that are used. Filament ships with 6 predefined colors that are used everywhere within the framework. They are customizable as follows:

```
use Filament\Panel;
use Filament\Support\Colors\Color;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->colors([
            'danger' => Color::Rose,
            'gray' => Color::Gray,
            'info' => Color::Blue,
            'primary' => Color::Indigo,
            'success' => Color::Emerald,
            'warning' => Color::Orange,
        ]);
}
```

The `Filament\Support\Colors\Color` class contains color options for all [Tailwind CSS color palettes](#).

You can also pass in a function to `register()` which will only get called when the app is getting rendered. This is useful if you are calling `register()` from a service provider, and want to access objects like the currently authenticated user, which are initialized later in middleware.

Alternatively, you may pass your own palette in as an array of RGB values:

```
$panel
->colors([
    'primary' => [
        50 => '238, 242, 255',
        100 => '224, 231, 255',
        200 => '199, 210, 254',
        300 => '165, 180, 252',
        400 => '129, 140, 248',
        500 => '99, 102, 241',
        600 => '79, 70, 229',
        700 => '67, 56, 202',
        800 => '55, 48, 163',
        900 => '49, 46, 129',
        950 => '30, 27, 75',
    ],
])
```

Generating a color palette

If you want us to attempt to generate a palette for you based on a singular hex or RGB value, you can pass that in:

```
$panel
->colors([
    'primary' => '#6366f1',
])
```

```
$panel
->colors([
    'primary' => 'rgb(99, 102, 241)',
])
```

Changing the font

By default, we use the [Inter](#) font. You can change this using the `font()` method in the [configuration](#) file:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->font('Poppins');
}
```

All [Google Fonts](#) are available to use.

Changing the font provider

[Bunny Fonts CDN](#) is used to serve the fonts. It is GDPR-compliant. If you'd like to use [Google Fonts CDN](#) instead, you can do so using the `provider` argument of the `font()` method:

```
use Filament\FontProviders\GoogleFontProvider;

$panel->font('Inter', provider: GoogleFontProvider::class)
```

Or if you'd like to serve the fonts from a local stylesheet, you can use the `LocalFontProvider`:

```
use Filament\FontProviders\LocalFontProvider;

$panel->font(
    'Inter',
    url: asset('css/fonts.css'),
    provider: LocalFontProvider::class,
)
```

Creating a custom theme

Filament allows you to change the CSS used to render the UI by compiling a custom stylesheet to replace the default one. This custom stylesheet is called a "theme".

Themes use [Tailwind CSS](#), the Tailwind Forms plugin, the Tailwind Typography plugin, the [PostCSS Nesting plugin](#), and [Autoprefixer](#).

To create a custom theme for a panel, you can use the `php artisan make:filament-theme` command:

```
php artisan make:filament-theme
```

If you have multiple panels, you can specify the panel you want to create a theme for:

```
php artisan make:filament-theme admin
```

By default, this command will use NPM to install dependencies. If you want to use a different package manager, you can use the `--pm` option:

```
php artisan make:filament-theme --pm=bun
```

The command will create a CSS file and Tailwind Configuration file in the `/resources/css/filament` directory. You can then customize the theme by editing these files. It will also give you instructions on how to compile the theme and register it in Filament. **Please follow the instructions in the command to complete the setup process:**

- ↳ First, add a `new` item to the `'input'` array of `'vite.config.js'`:
`'resources/css/filament/admin/theme.css'`
- ↳ Next, register the theme in the admin panel provider using `'-`
`>viteTheme('resources/css/filament/admin/theme.css')`
- ↳ Finally, run `'npm run build'` to compile the theme

Please reference the command to see the exact file names that you need to register, they may not be `admin/theme.css`.

Disabling dark mode

To disable dark mode switching, you can use the configuration file:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->darkMode(false);
}
```

Changing the default theme mode

By default, Filament uses the user's system theme as the default mode. For example, if the user's computer is in dark mode, Filament will use dark mode by default. The system mode in Filament is reactive if the user changes their computer's mode. If you want to change the default mode to force light or dark mode, you can use the

`defaultThemeMode()` method, passing `ThemeMode::Light` or `ThemeMode::Dark`:

```
use Filament\Enums\ThemeMode;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->defaultThemeMode(ThemeMode::Light);
}
```

Adding a logo

By default, Filament uses your app's name to render a simple text-based logo. However, you can easily customize this.

If you want to simply change the text that is used in the logo, you can use the `brandName()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->brandName('Filament Demo');
}
```

To render an image instead, you can pass a URL to the `brandLogo()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->brandLogo(asset('images/logo.svg'));
}
```

Alternatively, you may directly pass HTML to the `brandLogo()` method to render an inline SVG element for example:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->brandLogo(fn () => view('filament.admin.logo'));
}
```

```
<svg
    viewBox="0 0 128 26"
    xmlns="http://www.w3.org/2000/svg"
    class="h-full fill-gray-500 dark:fill-gray-400"
>
<!-- ... -->
</svg>
```

If you need a different logo to be used when the application is in dark mode, you can pass it to `darkModeBrandLogo()` in the same way.

The logo height defaults to a sensible value, but it's impossible to account for all possible aspect ratios. Therefore, you may customize the height of the rendered logo using the `brandLogoHeight()` method:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->brandLogo(fn () => view('filament.admin.logo'))
        ->brandLogoHeight('2rem');
}
```

Adding a favicon

To add a favicon, you can use the [configuration](#) file, passing the public URL of the favicon:

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->favicon(asset('images/favicon.png'));
}
```

Plugins

Overview

The basis of Filament plugins are Laravel packages. They are installed into your Filament project via Composer, and follow all the standard techniques, like using service providers to register routes, views, and translations. If you're new to Laravel package development, here are some resources that can help you grasp the core concepts:

- [The Package Development section of the Laravel docs](#) serves as a great reference guide.
- [Spatie's Package Training course](#) is a good instructional video series to teach you the process step by step.
- [Spatie's Package Tools](#) allows you to simplify your service provider classes using a fluent configuration object.

Filament plugins build on top of the concepts of Laravel packages and allow you to ship and consume reusable features for any Filament panel. They can be added to each panel one at a time, and are also configurable differently per-panel.

Configuring the panel with a plugin class

A plugin class is used to allow your package to interact with a panel [configuration](#) file. It's a simple PHP class that implements the `Plugin` interface. 3 methods are required:

- The `getId()` method returns the unique identifier of the plugin amongst other plugins. Please ensure that it is specific enough to not clash with other plugins that might be used in the same project.
- The `register()` method allows you to use any [configuration](#) option that is available to the panel. This includes registering [resources](#), [custom pages](#), [themes](#), [render hooks](#) and more.
- The `boot()` method is run only when the panel that the plugin is being registered to is actually in-use. It is executed by a middleware class.

```
<?php

namespace DanHarrin\FilamentBlog;

use DanHarrin\FilamentBlog\Pages\Settings;
use DanHarrin\FilamentBlog\Resources\CategoryResource;
use DanHarrin\FilamentBlog\Resources\PostResource;
use Filament\Contracts\Plugin;
use Filament\Panel;

class BlogPlugin implements Plugin
{
    public function getId(): string
    {
        return 'blog';
    }

    public function register(Panel $panel): void
    {
        $panel
            ->resources([
                PostResource::class,
                CategoryResource::class,
            ])
            ->pages([
                Settings::class,
            ]);
    }

    public function boot(Panel $panel): void
    {
        //
    }
}
```

The users of your plugin can add it to a panel by instantiating the plugin class and passing it to the `plugin()` method of the [configuration](#):

```
use DanHarrin\FilamentBlog\BlogPlugin;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->plugin(new BlogPlugin());
}
```

Fluently instantiating the plugin class

You may want to add a `make()` method to your plugin class to provide a fluent interface for your users to instantiate it. In addition, by using the container (`app()`) to instantiate the plugin object, it can be replaced with a different implementation at runtime:

```
use Filament\Contracts\Plugin;

class BlogPlugin implements Plugin
{
    public static function make(): static
    {
        return app(static::class);
    }

    // ...
}
```

Now, your users can use the `make()` method:

```
use DanHarrin\FilamentBlog\BlogPlugin;
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
    // ...
    ->plugin(BlogPlugin::make());
}
```

Configuring plugins per-panel

You may add other methods to your plugin class, which allow your users to configure it. We suggest that you add both a setter and a getter method for each option you provide. You should use a property to store the preference in the setter and retrieve it again in the getter:

```

use DanHarrin\FilamentBlog\Resources\AuthorResource;
use Filament\Contracts\Plugin;
use Filament\Panel;

class BlogPlugin implements Plugin
{
    protected bool $hasAuthorResource = false;

    public function authorResource(bool $condition = true): static
    {
        // This is the setter method, where the user's preference is
        // stored in a property on the plugin object.
        $this->hasAuthorResource = $condition;

        // The plugin object is returned from the setter method to
        // allow fluent chaining of configuration options.
        return $this;
    }

    public function hasAuthorResource(): bool
    {
        // This is the getter method, where the user's preference
        // is retrieved from the plugin property.
        return $this->hasAuthorResource;
    }

    public function register(Panel $panel): void
    {
        // Since the `register()` method is executed after the user
        // configures the plugin, you can access any of their
        // preferences inside it.
        if ($this->hasAuthorResource()) {
            // Here, we only register the author resource on the
            // panel if the user has requested it.
            $panel->resources([
                AuthorResource::class,
            ]);
        }
    }

    // ...
}

```

Additionally, you can use the unique ID of the plugin to access any of its configuration options from outside the plugin class. To do this, pass the ID to the `filament()` method:

```
filament('blog')->hasAuthorResource()
```

You may wish to have better type safety and IDE autocompletion when accessing configuration. It's completely up to you how you choose to achieve this, but one idea could be adding a static method to the plugin class to retrieve it:

```
use Filament\Contracts\Plugin;

class BlogPlugin implements Plugin
{
    public static function get(): static
    {
        return filament(app(static::class)->getId());
    }

    // ...
}
```

Now, you can access the plugin configuration using the new static method:

```
BlogPlugin::get()->hasAuthorResource()
```

Distributing a panel in a plugin

It's very easy to distribute an entire panel in a Laravel package. This way, a user can simply install your plugin and have an entirely new part of their app pre-built.

When configuring a panel, the configuration class extends the `PanelProvider` class, and that is a standard Laravel service provider. You can use it as a service provider in your package:

```
<?php

namespace DanHarrin\FilamentBlog;

use Filament\Panel;
use Filament\PanelProvider;

class BlogPanelProvider extends PanelProvider
{
    public function panel(Panel $panel): Panel
    {
        return $panel
            ->id('blog')
            ->path('blog')
            ->resources([
                // ...
            ])
            ->pages([
                // ...
            ])
            ->widgets([
                // ...
            ])
            ->middleware([
                // ...
            ])
            ->authMiddleware([
                // ...
            ]);
    }
}
```

You should then register it as a service provider in the `composer.json` of your package:

```
"extra": {
    "laravel": {
        "providers": [
            "DanHarrin\\FilamentBlog\\BlogPanelProvider"
        ]
    }
}
```

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Since all pages in the app are Livewire components, we're just using Livewire testing helpers everywhere. If you've never tested Livewire components before, please read [this guide](#) from the Livewire docs.

Getting started

Ensure that you are authenticated to access the app in your `TestCase`:

```
protected function setUp(): void
{
    parent::setUp();

    $this->actingAs(User::factory()->create());
}
```

Testing multiple panels

If you have multiple panels and you would like to test a non-default panel, you will need to tell Filament which panel you are testing. This can be done in the `setUp()` method of the test case, or you can do it at the start of a particular test. Filament usually does this in a middleware when you access the panel through a request, so if you're not making a request in your test like when testing a Livewire component, you need to set the current panel manually:

```
use Filament\Facades\Filament;

Filament::setCurrentPanel(
    Filament::getPanel('app'), // Where `app` is the ID of the panel you want to test.
);
```

Resources

Pages

List

Routing & render

To ensure that the List page for the `PostResource` is able to render successfully, generate a page URL, perform a request to this URL and ensure that it is successful:

```
it('can render page', function () {
    $this->get(PostResource::getUrl('index'))->assertSuccessful();
});
```

Table

Filament includes a selection of helpers for testing tables. A full guide to testing tables can be found in the [Table Builder documentation](#).

To use a table [testing helper](#), make assertions on the resource's List page class, which holds the table:

```
use function Pest\LiveWire\livewire;

it('can list posts', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts);
});
```

Create

Routing & render

To ensure that the Create page for the `PostResource` is able to render successfully, generate a page URL, perform a request to this URL and ensure that it is successful:

```
it('can render page', function () {
    $this->get(PostResource::getUrl('create'))->assertSuccessful();
});
```

Creating

You may check that data is correctly saved into the database by calling `fillForm()` with your form data, and then asserting that the database contains a matching record:

```
use function Pest\LiveWire\livewire;

it('can create', function () {
    $newData = Post::factory()->make();

    livewire(PostResource\Pages\CreatePost::class)
        ->fillForm([
            'author_id' => $newData->author->getKey(),
            'content' => $newData->content,
            'tags' => $newData->tags,
            'title' => $newData->title,
        ])
        ->call('create')
        ->assertHasNoFormErrors();

    $this->assertDatabaseHas(Post::class, [
        'author_id' => $newData->author->getKey(),
        'content' => $newData->content,
        'tags' => json_encode($newData->tags),
        'title' => $newData->title,
    ]);
});
```

Validation

Use `assertHasFormErrors()` to ensure that data is properly validated in a form:

```
use function Pest\LiveWire\liveWire;

it('can validate input', function () {
    liveWire(PostResource\Pages\CreatePost::class)
        ->fillForm([
            'title' => null,
        ])
        ->call('create')
        ->assertHasFormErrors(['title' => 'required']);
});
```

Edit

Routing & render

To ensure that the Edit page for the `PostResource` is able to render successfully, generate a page URL, perform a request to this URL and ensure that it is successful:

```
it('can render page', function () {
    $this->get(PostResource::getUrl('edit', [
        'record' => Post::factory()->create(),
    ]))->assertSuccessful();
});
```

Filling existing data

To check that the form is filled with the correct data from the database, you may `assertFormSet()` that the data in the form matches that of the record:

```
use function Pest\LiveWire\liveWire;

it('can retrieve data', function () {
    $post = Post::factory()->create();

    liveWire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
        ->assertFormSet([
            'author_id' => $post->author->getKey(),
            'content' => $post->content,
            'tags' => $post->tags,
            'title' => $post->title,
        ]);
});
```

Saving

You may check that data is correctly saved into the database by calling `fillForm()` with your form data, and then asserting that the database contains a matching record:

```
use function Pest\Livewire\livewire;

it('can save', function () {
    $post = Post::factory()->create();
    $newData = Post::factory()->make();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->fillForm([
        'author_id' => $newData->author->getKey(),
        'content' => $newData->content,
        'tags' => $newData->tags,
        'title' => $newData->title,
    ])
    ->call('save')
    ->assertHasNoFormErrors();

    expect($post->refresh())
        ->author_id->toBe($newData->author->getKey())
        ->content->toBe($newData->content)
        ->tags->toBe($newData->tags)
        ->title->toBe($newData->title);
});
});
```

Validation

Use `assertHasFormErrors()` to ensure that data is properly validated in a form:

```
use function Pest\Livewire\livewire;

it('can validate input', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->fillForm([
        'title' => null,
    ])
    ->call('save')
    ->assertHasFormErrors(['title' => 'required']);
});
});
```

Deleting

You can test the `DeleteAction` using `callAction()`:

```
use Filament\Actions\DeleteAction;
use function Pest\Livewire\livewire;

it('can delete', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->callAction(DeleteAction::class);

    $this->assertModelMissing($post);
});
```

You can ensure that a particular user is not able to see a `DeleteAction` using `assertActionHidden()`:

```
use Filament\Actions\DeleteAction;
use function Pest\Livewire\livewire;

it('can not delete', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->assertActionHidden(DeleteAction::class);
});
```

View

Routing & render

To ensure that the View page for the `PostResource` is able to render successfully, generate a page URL, perform a request to this URL and ensure that it is successful:

```
it('can render page', function () {
    $this->get(PostResource::getUrl('view', [
        'record' => Post::factory()->create(),
    ]))->assertSuccessful();
});
```

Filling existing data

To check that the form is filled with the correct data from the database, you may `assertFormSet()` that the data in the form matches that of the record:

```
use function Pest\Livewire\livewire;

it('can retrieve data', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ViewPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->assertFormSet([
        'author_id' => $post->author->getKey(),
        'content' => $post->content,
        'tags' => $post->tags,
        'title' => $post->title,
    ]);
});
```

Relation managers

Render

To ensure that a relation manager is able to render successfully, mount the Livewire component:

```
use App\Filament\Resources\CategoryResource\Pages>EditCategory;
use function Pest\Livewire\livewire;

it('can render relation manager', function () {
    $category = Category::factory()
        ->has(Post::factory()->count(10))
        ->create();

    livewire(CategoryResource\RelationManagers\PostsRelationManager::class, [
        'ownerRecord' => $category,
        'pageClass' => EditCategory::class,
    ])
    ->assertSuccessful();
});
```

Table

Filament includes a selection of helpers for testing tables. A full guide to testing tables can be found [in the Table Builder documentation](#).

To use a table [testing helper](#), make assertions on the relation manager class, which holds the table:

```
use App\Filament\Resources\CategoryResource\Pages>EditCategory;
use function Pest\Livewire\livewire;

it('can list posts', function () {
    $category = Category::factory()
        ->has(Post::factory()->count(10))
        ->create();

    livewire(CategoryResource\RelationManagers\PostsRelationManager::class, [
        'ownerRecord' => $category,
        'pageClass' => EditCategory::class,
    ])
        ->assertCanSeeTableRecords($category->posts);
});
```

Upgrade Guide

If you see anything missing from this guide, please do not hesitate to [make a pull request](#) to our repository! Any help is appreciated!

New requirements

- Laravel v10.0+
- Livewire v3.0+

Please upgrade Filament before upgrading to Livewire v3. Instructions on how to upgrade Livewire can be found [here](#).

Upgrading automatically

The easiest way to upgrade your app is to run the automated upgrade script. This script will automatically upgrade your application to the latest version of Filament and make changes to your code, which handles most breaking changes.

```
composer require filament/upgrade:^3.2 -W --dev
vendor/bin/filament-v3
```

Make sure to carefully follow the instructions, and review the changes made by the script. You may need to make some manual changes to your code afterwards, but the script should handle most of the repetitive work for you.

A new `app/Providers/Filament/*PanelProvider.php` file will be created, and the configuration from your old `config/filament.php` file should be copied. Since this is a [Laravel service provider](#), it needs to be registered in `config/app.php`. Filament will attempt to do this for you, but if you get an error while trying to access your panel, then this process has probably failed. You can manually register the service provider by adding it to the `providers` array.

Finally, you must run `php artisan filament:install` to finalize the Filament v3 installation. This command must be run for all new Filament projects.

You can now `composer remove filament/upgrade` as you don't need it anymore.

Some plugins you're using may not be available in v3 just yet. You could temporarily remove them from your `composer.json` file until they've been upgraded, replace them with a similar plugins that are v3-compatible, wait for the plugins to be upgraded before upgrading your app, or even write PRs to help the authors upgrade them.

Upgrading manually

After upgrading the dependency via Composer, you should execute `php artisan filament:upgrade` in order to clear any Laravel caches and publish the new frontend assets.

High-impact changes

Panel provider instead of the config file

The Filament v2 config file grew pretty big, and now it is incredibly small. Most of the configuration is now done in a service provider, which provides a cleaner API, more type safety, IDE autocomplete support, and [the ability to create multiple panels in your app](#). We call these special configuration service providers **"panel providers"**.

Before you can create the new panel provider, make sure that you've got Filament v3 installed with Composer. Then, run the following command:

```
php artisan filament:install --panels
```

A new `app/Providers/Filament/AdminPanelProvider.php` file will be created, ready for you to transfer over your old configuration from the `config/filament.php` file. Since this is a [Laravel service provider](#), it needs to be registered in `config/app.php`. Filament will attempt to do this for you, but if you get an error while trying to access your panel, then this process has probably failed. You can manually register the service provider by adding it to the `providers` array.

Most configuration transfer is very self-explanatory, but if you get stuck, please refer to the [configuration documentation](#).

This will especially affect configuration done via the `Filament::serving()` method, which was used for theme customization, navigation and menu registration. Consult the [configuration](#), [navigation](#) and [themes](#) documentation sections.

Finally, you can run the following command to replace the old config file with the shiny new one:

```
php artisan vendor:publish --tag=filament-config --force
```

FILAMENT_FILESYSTEM_DRIVER .env variable

The `FILAMENT_FILESYSTEM_DRIVER` .env variable has been renamed to `FILAMENT_FILESYSTEM_DISK`. This is to make it more consistent with Laravel, as Laravel v9 introduced this change as well. Please ensure that you update your .env files accordingly, and don't forget production!

Resource and relation manager imports

Some classes that are imported in resources and relation managers have moved:

- `Filament\Resources\Form` has moved to `Filament\Forms\Form`
- `Filament\Resources\Table` has moved to `Filament\Tables\Table`

Method signature changes

User model (with `FilamentUser` interface):

- `canAccessFilament()` has been renamed to `canAccessPanel()` and has a new `\Filament\Panel $panel` parameter

Resource classes:

- `applyGlobalSearchAttributeConstraint()` now has a `string $search` parameter before `$searchAttributes()` instead of `$searchQuery` after
- `getGlobalSearchEloquentQuery()` is public
- `getGlobalSearchResults()` has a `$search` parameter instead of `$searchQuery`
- `getRouteBaseName()` has a new `?string $panel` parameter

Resource classes and all page classes, including resource pages, custom pages, settings pages, and dashboard pages:

- `getActiveNavigationIcon()` is public
- `getNavigationBadge()` is public
- `getNavigationBadgeColor()` is public
- `getNavigationGroup()` is public
- `getNavigationIcon()` is public
- `getNavigationLabel()` is public
- `getNavigationSort()` is public
- `getNavigationUrl()` is public
- `shouldRegisterNavigation()` is public

All page classes, including resource pages, custom pages, settings pages, and custom dashboard pages:

- `getBreadcrumbs()` is public
- `getFooterWidgetsColumns()` is public
- `getHeader()` is public
- `getHeaderWidgetsColumns()` is public
- `getHeading()` is public
- `getRouteName()` has a new `?string $panel` parameter
- `getSubheading()` is public
- `getTitle()` is public
- `getVisibleFooterWidgets()` is public
- `getVisibleHeaderWidgets()` is public

List and Manage resource pages:

- `table()` is public

Create resource pages:

- `canCreateAnother()` is public

Edit and View resource pages:

- `getFormTabLabel()` is now `getContentTabLabel()`

Relation managers:

- `form()` is no longer static
- `getInverseRelationshipName()` return type is now `?string`
- `table()` is no longer static

Custom dashboard pages:

- `getDashboard()` is public
- `getWidgets()` is public

Property signature changes

Resource classes and all page classes, including resource pages, custom pages, settings pages, and dashboard pages:

- `$middlewares` is now `$routeMiddleware`

Heroicons have been updated to v2

The Heroicons library has been updated to v2. This means that any icons you use in your app may have changed names. You can find a list of changes [here](#).

Medium-impact changes

Date-time pickers

The date-time picker form field now uses the browser's native date picker by default. It usually has a better UX than the old date picker, but you may notice features missing, bad browser compatibility, or behavioral bugs. If you want to revert to the old date picker, you can use the `native(false)` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->native(false)
```

Secondary color

Filament v2 had a `secondary` color for many components which was gray. All references to `secondary` should be replaced with `gray` to preserve the same appearance. This frees `secondary` to be registered to a new custom color of your choice.

`$get` and `$set` closure parameters

In the Form Builder package, `$get` and `$set` parameters now use a type of either `\Filament\Forms\Get` or `\Filament\Forms\Set` instead of `\Closure`. This allows for better IDE autocomplete support of each function's parameters.

An easy way to upgrade your code quickly is to find and replace:

- `Closure $get` to `\Filament\Forms\Get $get`
- `Closure $set` to `\Filament\Forms\Set $set`

Blade icon components have been disabled

During v2, we noticed performance issues with Blade icon components. We've decided to disable them by default in v3, so we only use the `@svg()` syntax for rendering icons.

A side effect of this change is that all custom icons that you use must now be registered in a set. We no longer allow arbitrary Blade components to be used as custom icons.

Logo customization

In v2, you can customize the logo of the admin panel using a

`/resources/views/vendor/filament/components/brand.blade.php` file. In v3, this has been moved to the new `brandLogo()` API. You can now set the brand logo by adding it to your panel configuration.

Plugins

Filament v3 has a new universal plugin system that breaches the constraints of the admin panel. Learn how to build v3 plugins [here](#).

Low-impact changes

Default actions and type-specific relation manager classes

If you started the Filament project after v2.13, you can skip this section. Since then, new resources and relation managers have been generated with the new syntax.

Since v2.13, resources and relation managers now define actions within the `table()` method instead of them being assumed by default.

When using simple resources, remove the `CanCreateRecords`, `CanDeleteRecords`, `CanEditRecords`, and `CanViewRecords` traits from the Manage page.

We also deprecated type-specific relation manager classes. Any classes extending `BelongsToRelationManager`, `HasManyRelationManager`, `HasManyThroughRelationManager`, `MorphManyRelationManager`, or `MorphToManyRelationManager` should now extend `\Filament\Resources\RelationManagers\RelationManager`. You can also remove the `CanAssociateRecords`, `CanAttachRecords`, `CanCreateRecords`, `CanDeleteRecords`, `CanDetachRecords`, `CanDisassociateRecords`, `CanEditRecords`, and `CanViewRecords` traits from relation managers.

To learn more about v2.13 changes, read our [blog post](#).

Blade components

Some Blade components have been moved to different namespaces:

- `<x-filament::page>` is now `<x-filament-panels::page>`

- `<x-filament::widget>` is now `<x-filament-widgets::widget>`

However, aliases have been set up so that you don't need to change your code.

Resource pages without a `$resource` property

Filament v2 allowed for resource pages to be created without a `$resource` property. In v3 you must declare this, else you may end up with the error:

```
Typed static property Filament\Resources\Pages\Page::$resource must not be accessed before  
initialization
```

You should ensure that the `$resource` property is set on all resource pages:

```
protected static string $resource = PostResource::class;
```

Chapter 2

Form Builder

Installation

The Form Builder package is pre-installed with the [Panel Builder](#). This guide is for using the Form Builder in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

Require the Form Builder package using Composer:

```
composer require filament/forms:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --forms  
npm install  
npm run dev
```

Existing Laravel projects

Run the following command to install the Form Builder assets:

```
php artisan filament:install --forms
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting  
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament `preset` (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Upgrading from Filament v2? Please review the [upgrade guide](#).

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post/autoload/dump` hook, you can run the command yourself:

```
composer update  
php artisan filament:upgrade
```

Getting Started

Overview

Filament's form package allows you to easily build dynamic forms in your app. You can use it to [add a form to any Livewire component](#). Additionally, it's used within other Filament packages to render forms within [app resources](#), [action modals](#), [table filters](#), and more. Learning how to build forms is essential to learning how to use these Filament packages.

This guide will walk you through the basics of building forms with Filament's form package. If you're planning to add a new form to your own Livewire component, you should [do that first](#) and then come back. If you're adding a form to an [app resource](#), or another Filament package, you're ready to go!

Form schemas

All Filament forms have a "schema". This is an array, which contains [fields](#) and [layout components](#).

Fields are the inputs that your user will fill their data into. For example, HTML's `<input>` or `<select>` elements. Each field has its own PHP class. For example, the `TextInput` class is used to render a text input field, and the `Select` class is used to render a select field. You can see a full [list of available fields here](#).

Layout components are used to group fields together, and to control how they are displayed. For example, you can use a `Grid` component to display multiple fields side-by-side, or a `Wizard` to separate fields into a multistep form. You can deeply nest layout components within each other to create very complex responsive UIs. You can see a full [list of available layout components here](#).

Adding fields to a form schema

Initialise a field or layout component with the `make()` method, and build a schema array with multiple fields:

```
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            TextInput::make('title'),
            TextInput::make('slug'),
            RichEditor::make('content'),
        ]);
}
```

Title**Slug****Content**

B I U S ☀ Heading Subheading “ “ </

Forms within a panel and other packages usually have 2 columns by default. For custom forms, you can use the `columns()` method to achieve the same effect:

```
$form
->schema([
    // ...
])
->columns(2);
```

Title	Slug
<input type="text"/>	<input type="text"/>
Content	
<input type="text" value="B I U S ⌂"/>	

Now, the `RichEditor` will only consume half of the available width. We can use the `columnSpan()` method to make it span the full width:

```
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\TextInput;

[
    TextInput::make('title'),
    TextInput::make('slug'),
    RichEditor::make('content')
        ->columnSpan(2), // or `columnSpan('full')`
]
```

The screenshot displays a form builder interface with the following components:

- Title:** A text input field.
- Slug:** A text input field.
- Content:** A rich text editor with a toolbar containing the following icons: B (bold), I (italic), U (underline), S (strikethrough), a link icon (🔗), Heading, Subheading, and code (‘‘ </’’).

You can learn more about columns and spans in the [layout documentation](#). You can even make them responsive!

Adding layout components to a form schema

Let's add a new `Section` to our form. `Section` is a layout component, and it allows you to add a heading and description to a set of fields. It can also allow fields inside it to collapse, which saves space in long forms.

```

use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\Section;
use Filament\Forms\Components\TextInput;

[
    TextInput::make('title'),
    TextInput::make('slug'),
    RichEditor::make('content')
        ->columnSpan(2),
    Section::make('Publishing')
        ->description('Settings for publishing this post.')
        ->schema([
            // ...
        ]),
]

```

In this example, you can see how the `Section` component has its own `schema()` method. You can use this to nest other fields and layout components inside:

```

use Filament\Forms\Components\DateTimePicker;
use Filament\Forms\Components\Section;
use Filament\Forms\Components\Select;

Section::make('Publishing')
    ->description('Settings for publishing this post.')
    ->schema([
        Select::make('status')
            ->options([
                'draft' => 'Draft',
                'reviewing' => 'Reviewing',
                'published' => 'Published',
            ]),
        DateTimePicker::make('published_at'),
    ])

```

Title**Slug****Content****B****I****U****S****Heading****Subheading****“****/<**

Publishing

Settings for publishing this post.

Status



Published at



This section now contains a `Select` field and a `DateTimePicker` field. You can learn more about those fields and their functionalities on the respective docs pages.

Validating fields

In Laravel, validation rules are usually defined in arrays like `['required', 'max:255']` or a combined string like `required|max:255`. This is fine if you're exclusively working in the backend with simple form requests. But Filament is also able to give your users frontend validation, so they can fix their mistakes before any backend requests are made.

In Filament, you can add validation rules to your fields by using methods like `required()` and `maxLength()`. This is also advantageous over Laravel's validation syntax, since your IDE can autocomplete these methods:

```
use Filament\Forms\Components\DateTimePicker;
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\Section;
use Filament\Forms\Components\Select;
use Filament\Forms\Components\TextInput;

[
    TextInput::make('title')
        ->required()
        ->maxLength(255),
    TextInput::make('slug')
        ->required()
        ->maxLength(255),
    RichEditor::make('content')
        ->columnSpan(2)
        ->maxLength(65535),
    Section::make('Publishing')
        ->description('Settings for publishing this post.')
        ->schema([
            Select::make('status')
                ->options([
                    'draft' => 'Draft',
                    'reviewing' => 'Reviewing',
                    'published' => 'Published',
                ])
                ->required(),
            DateTimePicker::make('published_at'),
        ]),
]
```

In this example, some fields are `required()`, and some have a `maxLength()`. We have [methods for most of Laravel's validation rules](#), and you can even add your own [custom rules](#).

Dependant fields

Since all Filament forms are built on top of Livewire, form schemas are completely dynamic. There are so many possibilities, but here are a couple of examples of how you can use this to your advantage:

Fields can hide or show based on another field's values. In our form, we can hide the `published_at` timestamp field until the `status` field is set to `published`. This is done by passing a closure to the `hidden()` method, which allows you to dynamically hide or show a field while the form is being used. Closures have access to many useful arguments like `$get`, and you can find a [full list here](#). The field that you depend on (the `status` in this case) needs to be set to `live()`, which tells the form to reload the schema each time it gets changed.

```

use Filament\Forms\Components\DateTimePicker;
use Filament\Forms\Components>Select;
use Filament\Forms\Get;

[
    Select::make('status')
        ->options([
            'draft' => 'Draft',
            'reviewing' => 'Reviewing',
            'published' => 'Published',
        ])
        ->required()
        ->live(),
    DateTimePicker::make('published_at')
        ->hidden(fn (Get $get) => $get('status') !== 'published'),
]

```

It's not just `hidden()` - all Filament form methods support closures like this. You can use them to change the label, placeholder, or even the options of a field, based on another. You can even use them to add new fields to the form, or remove them. This is a powerful tool that allows you to create complex forms with minimal effort.

Fields can also write data to other fields. For example, we can set the title to automatically generate a slug when the title is changed. This is done by passing a closure to the `afterStateUpdated()` method, which gets run each time the title is changed. This closure has access to the title (`$state`) and a function (`$set`) to set the slug field's state. You can find a [full list of closure arguments here](#). The field that you depend on (the `title` in this case) needs to be set to `live()`, which tells the form to reload and set the slug each time it gets changed.

```

use Filament\Forms\Components\TextInput;
use Filament\Forms\Set;
use Illuminate\Support\Str;

[
    TextInput::make('title')
        ->required()
        ->maxLength(255)
        ->live()
        ->afterStateUpdated(function (Set $set, $state) {
            $set('slug', Str::slug($state));
        }),
    TextInput::make('slug')
        ->required()
        ->maxLength(255),
]

```

Next steps with the forms package

Now you've finished reading this guide, where to next? Here are some suggestions:

- [Explore the available fields to collect input from your users.](#)
- [Check out the list of layout components to craft intuitive form structures with.](#)
- [Find out about all advanced techniques that you can customize forms to your needs.](#)
- [Write automated tests for your forms using our suite of helper methods.](#)

Fields

Getting Started

Overview

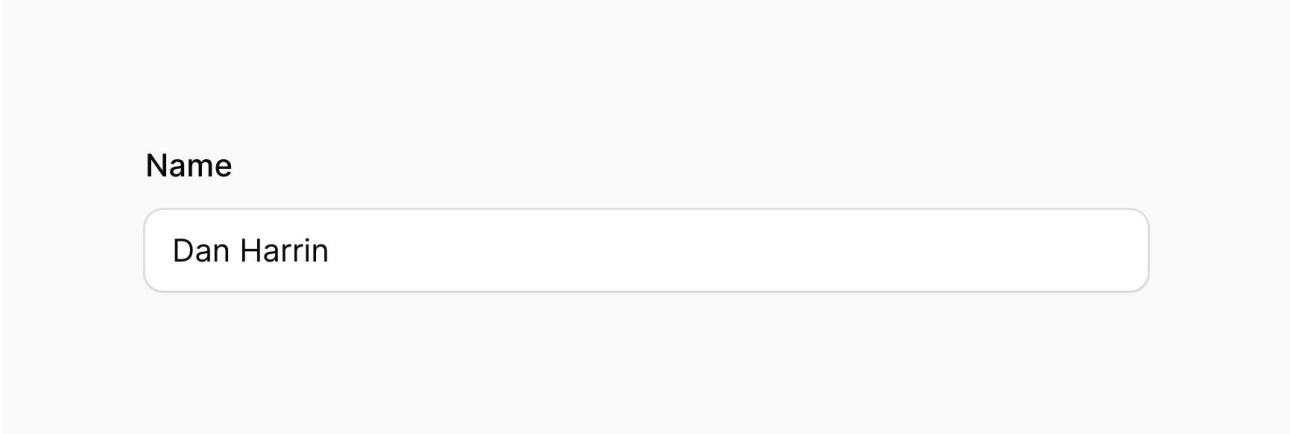
Field classes can be found in the `Filament\Forms\Components` namespace.

Fields reside within the schema of your form, alongside any [layout components](#).

Fields may be created using the static `make()` method, passing its unique name. The name of the field should correspond to a property on your Livewire component. You may use "dot notation" to bind fields to keys in arrays.

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
```



Name

Dan Harrin

Available fields

Filament ships with many types of field, suitable for editing different types of data:

- [Text input](#)
- [Select](#)
- [Checkbox](#)
- [Toggle](#)
- [Checkbox list](#)
- [Radio](#)
- [Date-time picker](#)
- [File upload](#)
- [Rich editor](#)
- [Markdown editor](#)
- [Repeater](#)
- [Builder](#)
- [Tags input](#)
- [Textarea](#)

- [Key-value](#)
- [Color picker](#)
- [Toggle buttons](#)
- [Hidden](#)

You may also [create your own custom fields](#) to edit data however you wish.

Setting a label

By default, the label of the field will be automatically determined based on its name. To override the field's label, you may use the `label()` method. Customizing the label in this way is useful if you wish to use a [translation string for localization](#):

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->label(__('fields.name'))
```

Optionally, you can have the label automatically translated [using Laravel's localization features](#) with the `translateLabel()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->translateLabel() // Equivalent to `label(__('Name'))`
```

Setting an ID

In the same way as labels, field IDs are also automatically determined based on their names. To override a field ID, use the `id()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->id('name-field')
```

Setting a default value

Fields may have a default value. This will be filled if the [form's `fill\(\)` method](#) is called without any arguments. To define a default value, use the `default()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->default('John')
```

Note that these defaults are only used when the form is loaded without existing data. Inside [panel resources](#) this only works on Create Pages, as Edit Pages will always fill the data from the model.

Adding helper text below the field

Sometimes, you may wish to provide extra information for the user of the form. For this purpose, you may add helper text below the field.

The `helperText()` method is used to add helper text:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->helperText('Your full name here, including any middle names.')
```

This method accepts a plain text string, or an instance of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even markdown, in the helper text:

```
use Filament\Forms\Components\TextInput;
use Illuminate\Support\HtmlString;

TextInput::make('name')
    ->helperText(new HtmlString('Your <strong>full name</strong> here, including any middle
names.'))

TextInput::make('name')
    ->helperText(str('Your **full name** here, including any middle names.')->inlineMarkdown()-
>toHtmlString())

TextInput::make('name')
    ->helperText(view('name-helper-text'))
```

Name

Dan Harrin

Your **full name** here, including any middle names.

Adding a hint next to the label

As well as `helperText` below the field, you may also add a "hint" next to the label of the field. This is useful for displaying additional information about the field, such as a link to a help page.

The `hint()` method is used to add a hint:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
    ->hint('Forgotten your password? Bad luck.')
```

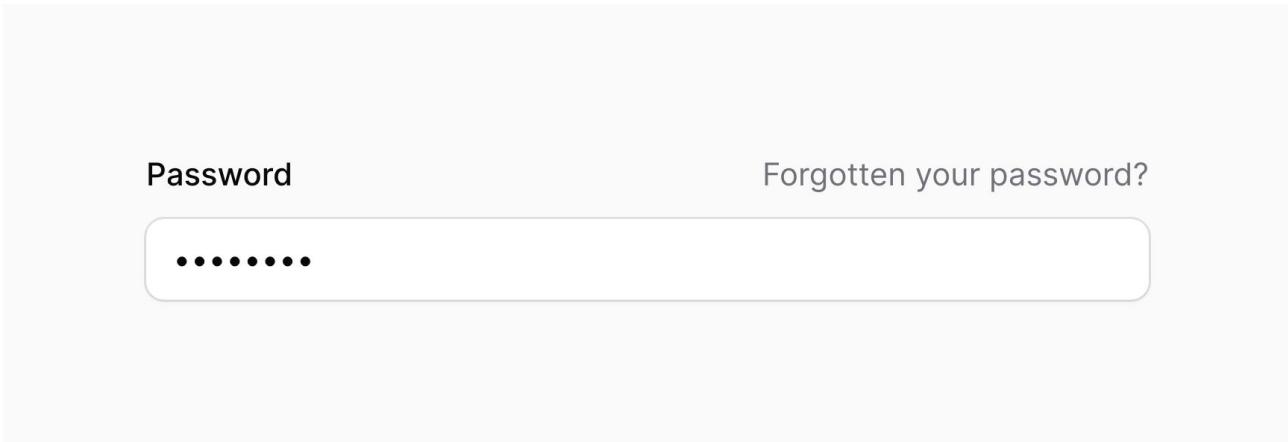
This method accepts a plain text string, or an instance of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even markdown, in the helper text:

```
use Filament\Forms\Components\TextInput;
use Illuminate\Support\HtmlString;

TextInput::make('password')
    ->hint(new HtmlString('<a href="/forgotten-password">Forgotten your password?</a>'))

TextInput::make('password')
    ->hint(str('[Forgotten your password?] (/forgotten-password)')->inlineMarkdown()-
        >toHtmlString())

TextInput::make('password')
    ->hint(view('forgotten-password-hint'))
```

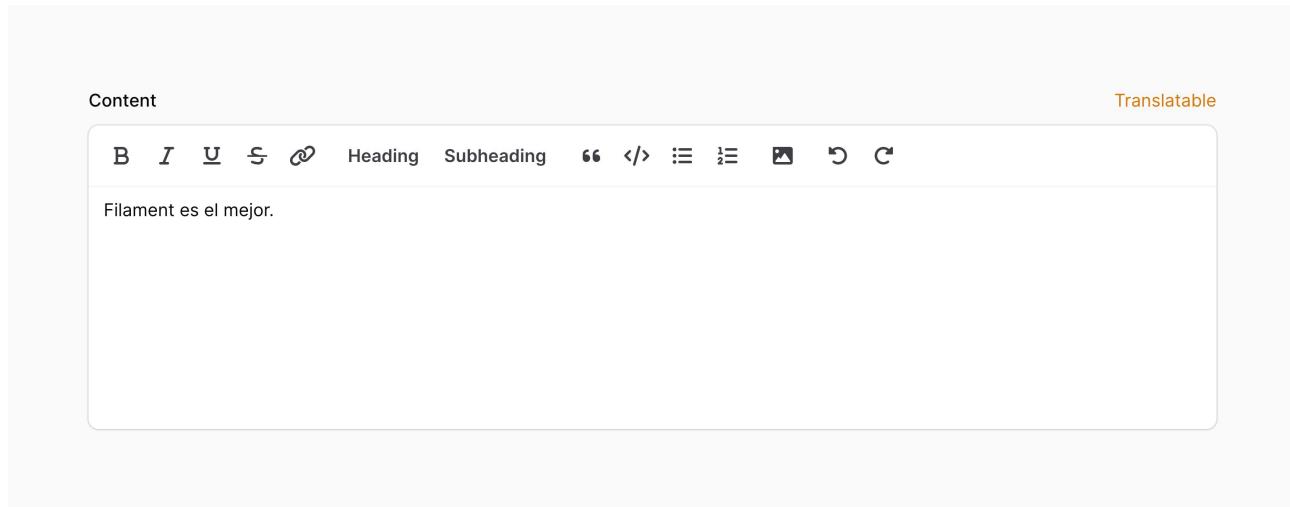


Changing the text color of the hint

You can change the text color of the hint. By default, it's gray, but you may use `danger`, `info`, `primary`, `success` and `warning`:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->hint('Translatable')
    ->hintColor('primary')
```

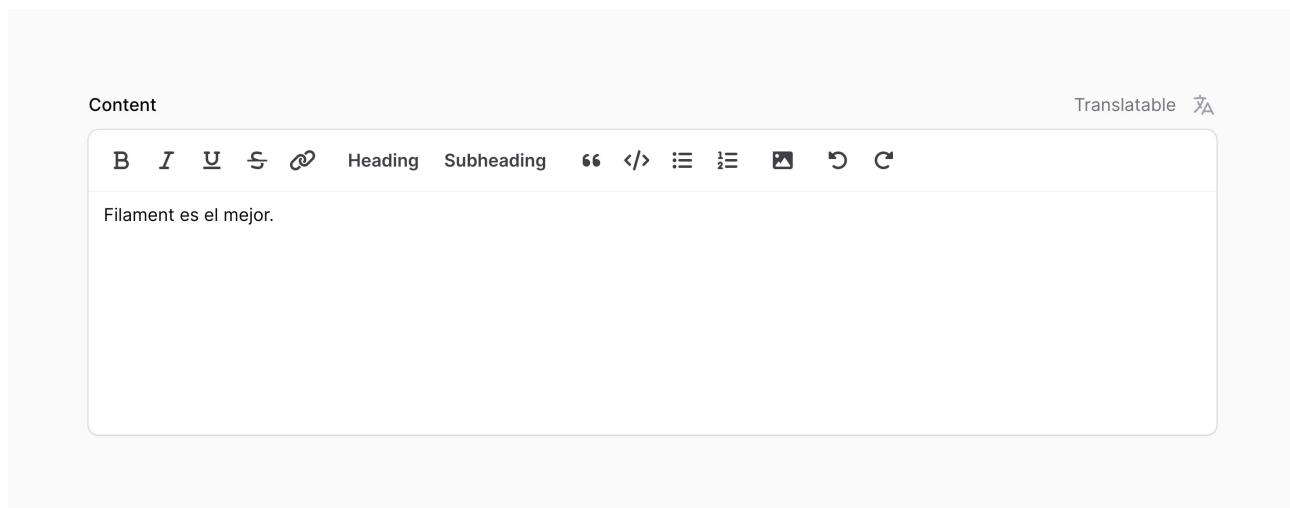


Adding an icon aside the hint

Hints may also have an `icon` rendered next to them:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->hint('Translatable')
    ->hintIcon('heroicon-m-language')
```



Adding a tooltip to a hint icon

Additionally, you can add a tooltip to display when you hover over the hint icon, using the `tooltip` parameter of `hintIcon()`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->hintIcon('heroicon-m-question-mark-circle', tooltip: 'Need some more information?')
```

Adding extra HTML attributes

You can pass extra HTML attributes to the field, which will be merged onto the outer DOM element. Pass an array of attributes to the `extraAttributes()` method, where the key is the attribute name and the value is the attribute value:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->extraAttributes(['title' => 'Text input'])
```

Some fields use an underlying `<input>` or `<select>` DOM element, but this is often not the outer element in the field, so the `extraAttributes()` method may not work as you wish. In this case, you may use the `extraInputAttributes()` method, which will merge the attributes onto the `<input>` or `<select>` element:

```
use Filament\Forms\Components\TextInput;

TextInput::make('categories')
->extraInputAttributes(['width' => 200])
```

You can also pass extra HTML attributes to the field wrapper which surrounds the label, entry, and any other text:

```
use Filament\Forms\Components\TextInput;

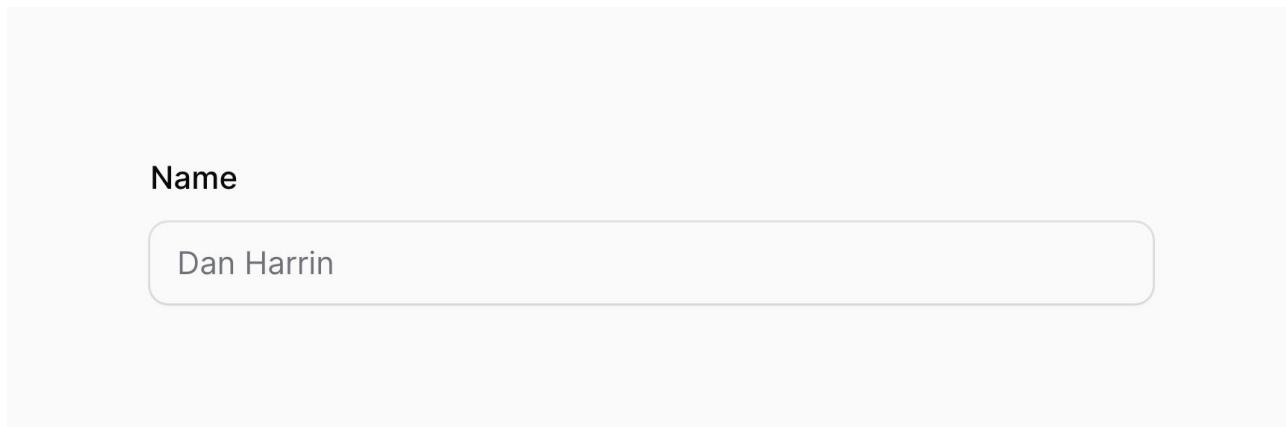
TextInput::make('categories')
->extraFieldWrapperAttributes(['class' => 'components-locked'])
```

Disabling a field

You may disable a field to prevent it from being edited by the user:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->disabled()
```



Name

Dan Harrin

Optionally, you may pass a boolean value to control if the field should be disabled or not:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
    ->disabled(! auth()->user()->isAdmin())
```

Disabling a field will prevent it from being saved. If you'd like it to be saved, but still not editable, use the `dehydrated()` method:

```
Toggle::make('is_admin')
    ->disabled()
    ->dehydrated()
```

If you choose to dehydrate the field, a skilled user could still edit the field's value by manipulating Livewire's JavaScript.

Hiding a field

You may hide a field:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->hidden()
```

Optionally, you may pass a boolean value to control if the field should be hidden or not:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->hidden(! auth()->user()->isAdmin())
```

Autofocusing a field when the form is loaded

Most fields are autofocusable. Typically, you should aim for the first significant field in your form to be autofocus for the best user experience. You can nominate a field to be autofocus using the `autofocus()` method:

```
use Filament\Forms\Components\TextInput;

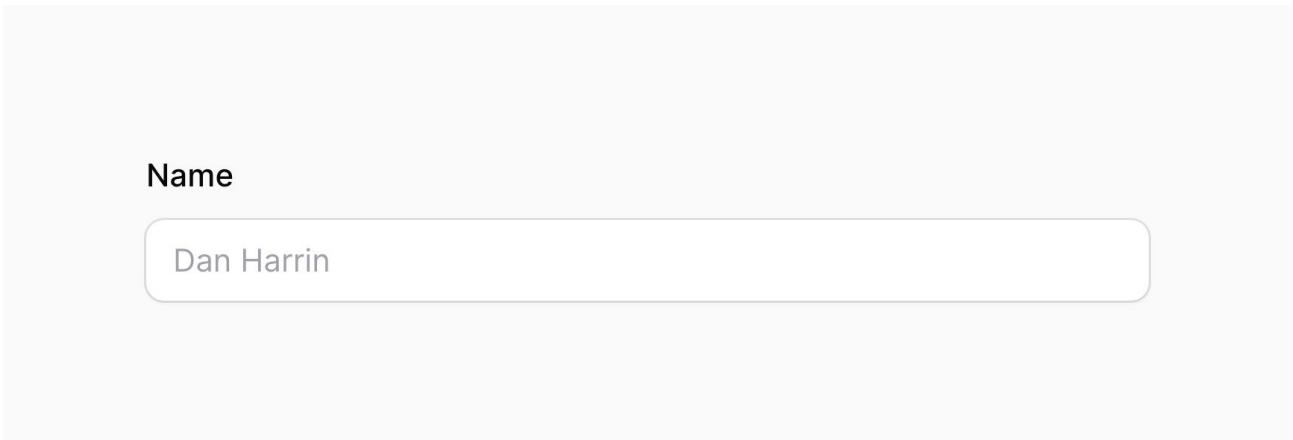
TextInput::make('name')
    ->autofocus()
```

Setting a placeholder

Many fields will also include a placeholder value for when it has no value. This is displayed in the UI but not saved if the field is submitted with no value. You may customize this placeholder using the `placeholder()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->placeholder('John Doe')
```



Marking a field as required

By default, required fields will show an asterisk next to their label. You may want to hide the asterisk on forms where all fields are required, or where it makes sense to add a hint to optional fields instead:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->required() // Adds validation to ensure the field is required
    ->markAsRequired(false) // Removes the asterisk
```

If your field is not `required()`, but you still wish to show an asterisk you can use `markAsRequired()` too:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->markAsRequired()
```

Global settings

If you wish to change the default behavior of a field globally, then you can call the static `configureUsing()` method inside a service provider's `boot()` method or a middleware. Pass a closure which is able to modify the component. For example, if you wish to make all checkboxes `inline(false)`, you can do it like so:

```
use Filament\Forms\Components\Checkbox;

Checkbox::configureUsing(function (Checkbox $checkbox): void {
    $checkbox->inline(false);
});
```

Of course, you are still able to overwrite this behavior on each field individually:

```
use Filament\Forms\Components\Checkbox;

Checkbox::make('is_admin')
    ->inline()
```

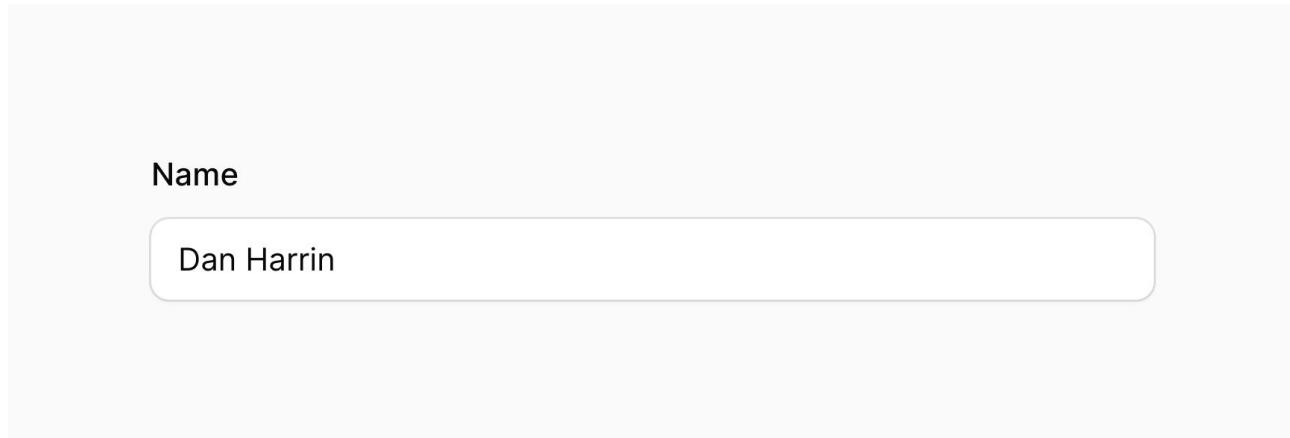
Text Input

Overview

The text input allows you to interact with a string:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
```



Setting the HTML input type

You may set the type of string using a set of methods. Some, such as `email()`, also provide validation:

```
use Filament\Forms\Components\TextInput;

TextInput::make('text')
    ->email() // or
    ->numeric() // or
    ->integer() // or
    ->password() // or
    ->tel() // or
    ->url()
```

You may instead use the `type()` method to pass another [HTML input type](#):

```
use Filament\Forms\Components\TextInput;

TextInput::make('backgroundColor')
    ->type('color')
```

Setting the HTML input mode

You may set the [`inputmode`](#) attribute of the input using the `inputMode()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('text')
->numeric()
->inputMode('decimal')
```

Setting the numeric step

You may set the `step` attribute of the input using the `step()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('number')
->numeric()
->step(100)
```

Autocompleting text

You may allow the text to be autocompleted by the browser using the `autocomplete()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
->password()
->autocomplete('new-password')
```

As a shortcut for `autocomplete="off"`, you may use `autocomplete(false)`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
->password()
->autocomplete(false)
```

For more complex autocomplete options, text inputs also support datalists.

Autocompleting text with a datalist

You may specify datalist options for a text input using the `datalist()` method:

```
TextInput::make('manufacturer')
->datalist([
    'BMW',
    'Ford',
    'Mercedes-Benz',
    'Porsche',
    'Toyota',
    'Tesla',
    'Volkswagen',
])
```

Datalists provide autocomplete options to users when they use a text input. However, these are purely recommendations, and the user is still able to type any value into the input. If you're looking to strictly limit users to a set of predefined options, check out the [select field](#).

Autocapitalizing text

You may allow the text to be [autocapitalized by the browser](#) using the `autocapitalize()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->autocapitalize('words')
```

Adding affix text aside the field

You may place text before and after the input using the `prefix()` and `suffix()` methods:

```
use Filament\Forms\Components\TextInput;

TextInput::make('domain')
->prefix('https://')
->suffix('.com')
```

Domain

https://	filamentphp	.com
----------	-------------	------

Using icons as affixes

You may place an [icon](#) before and after the input using the `prefixIcon()` and `suffixIcon()` methods:

```
use Filament\Forms\Components\TextInput;

TextInput::make('domain')
->url()
->suffixIcon('heroicon-m-globe-alt')
```

Domain

<https://filamentphp.com>



Setting the affix icon's color

Affix icons are gray by default, but you may set a different color using the `prefixIconColor()` and `suffixIconColor()` methods:

```
use Filament\Forms\Components\TextInput;

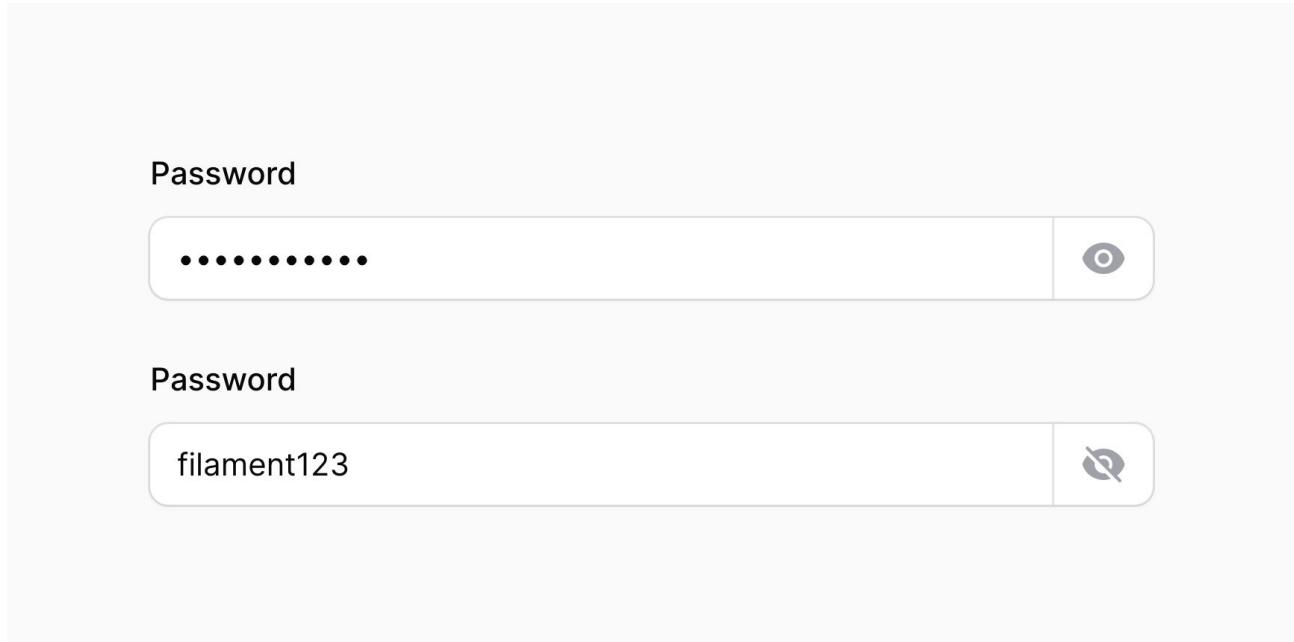
TextInput::make('domain')
    ->url()
    ->suffixIcon('heroicon-m-check-circle')
    ->suffixIconColor('success')
```

Revealable password inputs

When using `password()`, you can also make the input `revealable()`, so that the user can see a plain text version of the password they're typing by clicking a button:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
    ->password()
    ->revealable()
```



Input masking

Input masking is the practice of defining a format that the input value must conform to.

In Filament, you may use the `mask()` method to configure an [Alpine.js mask](#):

```
use Filament\Forms\Components\TextInput;

TextInput::make('birthday')
    ->mask('99/99/9999')
    ->placeholder('MM/DD/YYYY')
```

To use a [dynamic mask](#), wrap the JavaScript in a `RawJs` object:

```
use Filament\Forms\Components\TextInput;
use Filament\Support\RawJs;

TextInput::make('cardNumber')
    ->mask(RawJs::make(<<< 'JS'
        $input.startsWith('34') || $input.startsWith('37') ? '9999 999999 99999' : '9999 9999
9999 9999'
    JS))
```

Alpine.js will send the entire masked value to the server, so you may need to strip certain characters from the state before validating the field and saving it. You can do this with the `stripCharacters()` method, passing in a character or an array of characters to remove from the masked value:

```
use Filament\Forms\Components\TextInput;
use Filament\Support\RawJs;

TextInput::make('amount')
->mask(RawJs::make('$money($input)'))
->stripCharacters(',')
->numeric()
```

Making the field read-only

Not to be confused with [disabling the field](#), you may make the field "read-only" using the `readOnly()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->readOnly()
```

There are a few differences, compared to [disabled\(\)](#):

- When using `readOnly()`, the field will still be sent to the server when the form is submitted. It can be mutated with the browser console, or via JavaScript. You can use `dehydrated(false)` to prevent this.
- There are no styling changes, such as less opacity, when using `readOnly()`.
- The field is still focusable when using `readOnly()`.

Text input validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to text inputs.

Length validation

You may limit the length of the input by setting the `minLength()` and `maxLength()` methods. These methods add both frontend and backend validation:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->minLength(2)
->maxLength(255)
```

You can also specify the exact length of the input by setting the `length()`. This method adds both frontend and backend validation:

```
use Filament\Forms\Components\TextInput;

TextInput::make('code')
->length(8)
```

Size validation

You may validate the minimum and maximum value of a numeric input by setting the `minValue()` and `maxValue()` methods:

```
use Filament\Forms\Components\TextInput;

TextInput::make('number')
->numeric()
->minValue(1)
->maxValue(100)
```

Phone number validation

When using a `tel()` field, the value will be validated using: `/^[\+]*[()]{0,1}[0-9]{1,4}[]]{0,1}[-\s\.\/0-9]*$/`.

If you wish to change that, then you can use the `telRegex()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('phone')
->tel()
->telRegex('/^[\+]*[()]{0,1}[0-9]{1,4}[]]{0,1}[-\s\.\/0-9]*$/')
```

Alternatively, to customize the `telRegex()` across all fields, use a service provider:

```
use Filament\Forms\Components\TextInput;

TextInput::configureUsing(function (TextInput $component): void {
    $component->telRegex('/^[\+]*[()]{0,1}[0-9]{1,4}[]]{0,1}[-\s\.\/0-9]*$/');
});
```

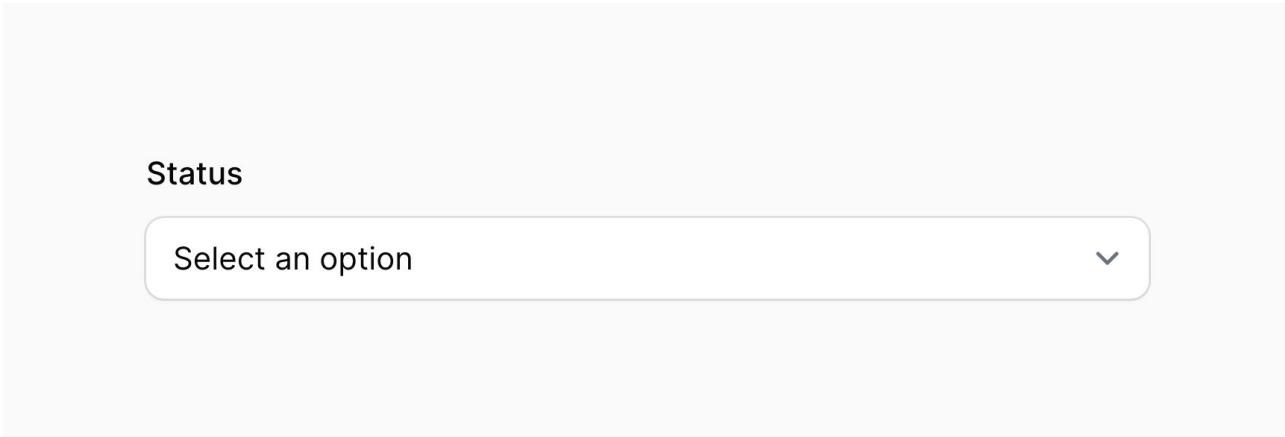
Select

Overview

The select component allows you to select from a list of predefined options:

```
use Filament\Forms\Components>Select;

Select::make('status')
->options([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
```

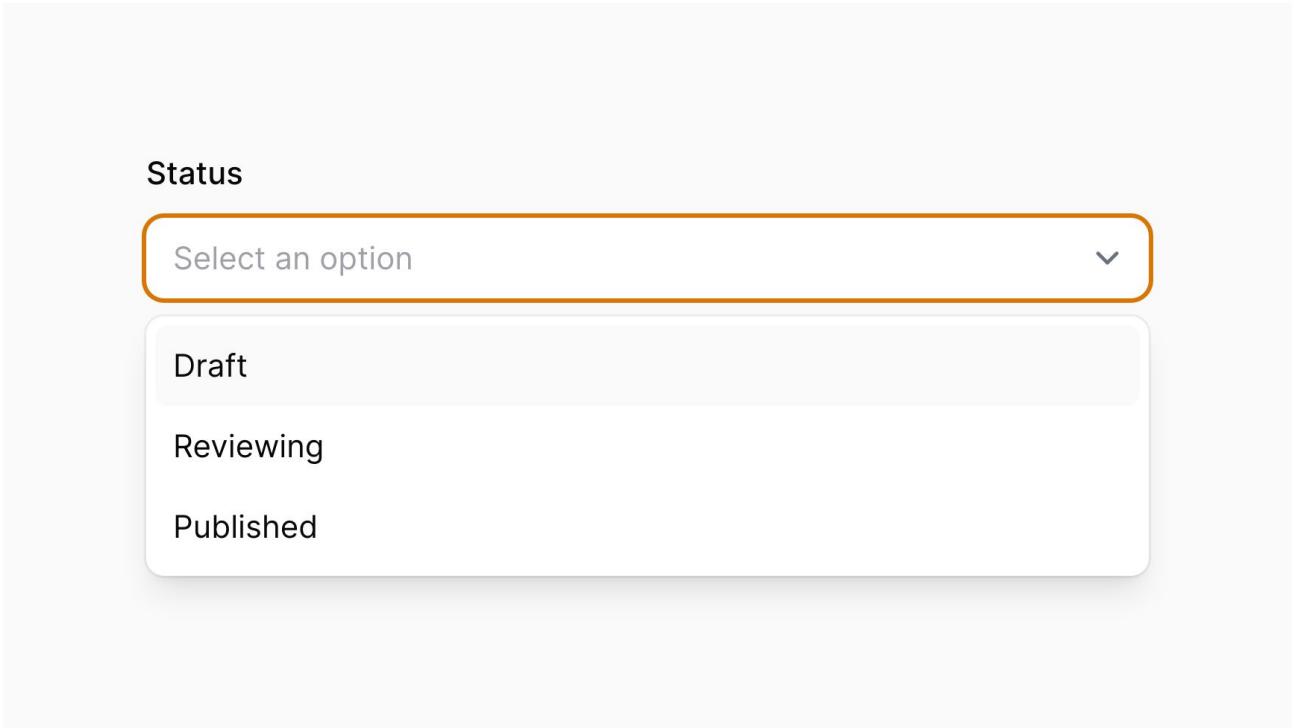


Enabling the JavaScript select

By default, Filament uses the native HTML5 select. You may enable a more customizable JavaScript select using the `native(false)` method:

```
use Filament\Forms\Components>Select;

Select::make('status')
->options([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
->native(false)
```

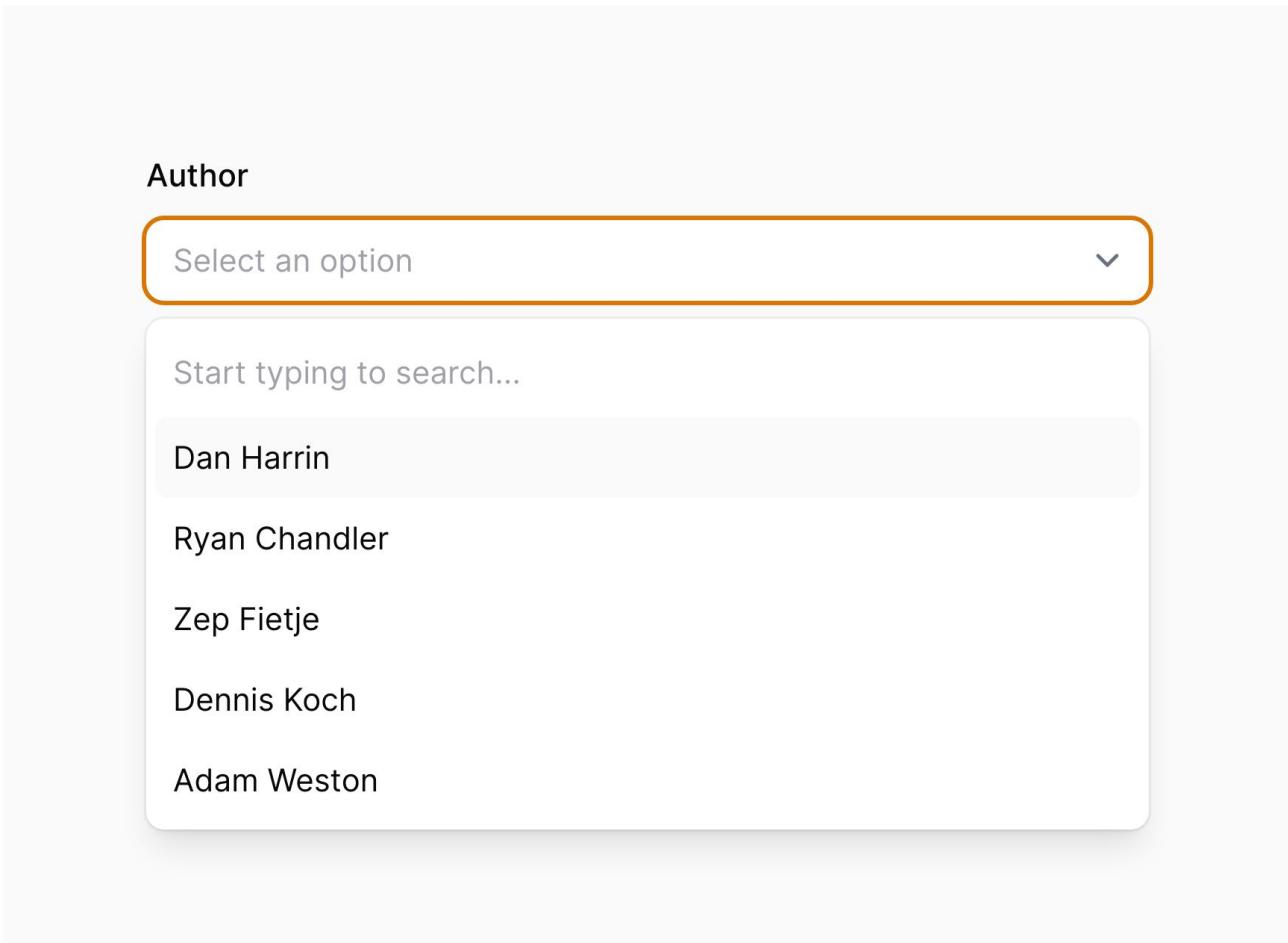


Searching options

You may enable a search input to allow easier access to many options, using the `searchable()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
->label('Author')
->options(User::all()->pluck('name', 'id'))
->searchable()
```



Returning custom search results

If you have lots of options and want to populate them based on a database search or other external data source, you can use the `getSearchResultsUsing()` and `getOptionLabelUsing()` methods instead of `options()`.

The `getSearchResultsUsing()` method accepts a callback that returns search results in `$key => $value` format. The current user's search is available as `$search`, and you should use that to filter your results.

The `getOptionLabelUsing()` method accepts a callback that transforms the selected option `$value` into a label. This is used when the form is first loaded when the user has not made a search yet. Otherwise, the label used to display the currently selected option would not be available.

Both `getSearchResultsUsing()` and `getOptionLabelUsing()` must be used on the select if you want to provide custom search results:

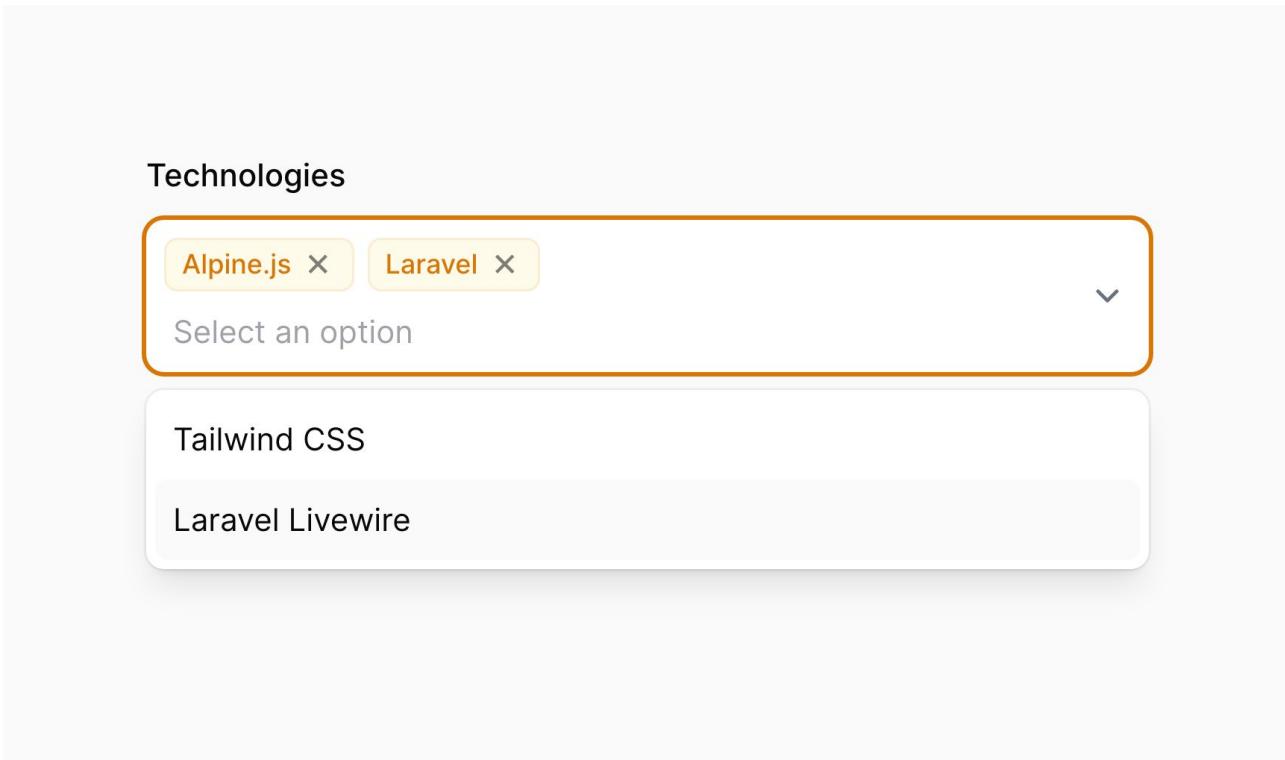
```
Select::make('author_id')
    ->searchable()
    ->getSearchResultsUsing(fn (string $search): array => User::where('name', 'like', "%{$search}%")->limit(50)->pluck('name', 'id')->toArray())
    ->getOptionLabelUsing(fn ($value): ?string => User::find($value)?->name),
```

Multi-select

The `multiple()` method on the `Select` component allows you to select multiple values from the list of options:

```
use Filament\Forms\Components>Select;

Select::make('technologies')
    ->multiple()
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
])
```



These options are returned in JSON format. If you're saving them using Eloquent, you should be sure to add an `array` cast to the model property:

```
use Illuminate\Database\Eloquent\Model;

class App extends Model
{
    protected $casts = [
        'technologies' => 'array',
    ];

    // ...
}
```

If you're returning custom search results, you should define `getOptionLabelsUsing()` instead of `getOptionLabelUsing()`. `$values` will be passed into the callback instead of `$value`, and you should return a `$key => $value` array of labels and their corresponding values:

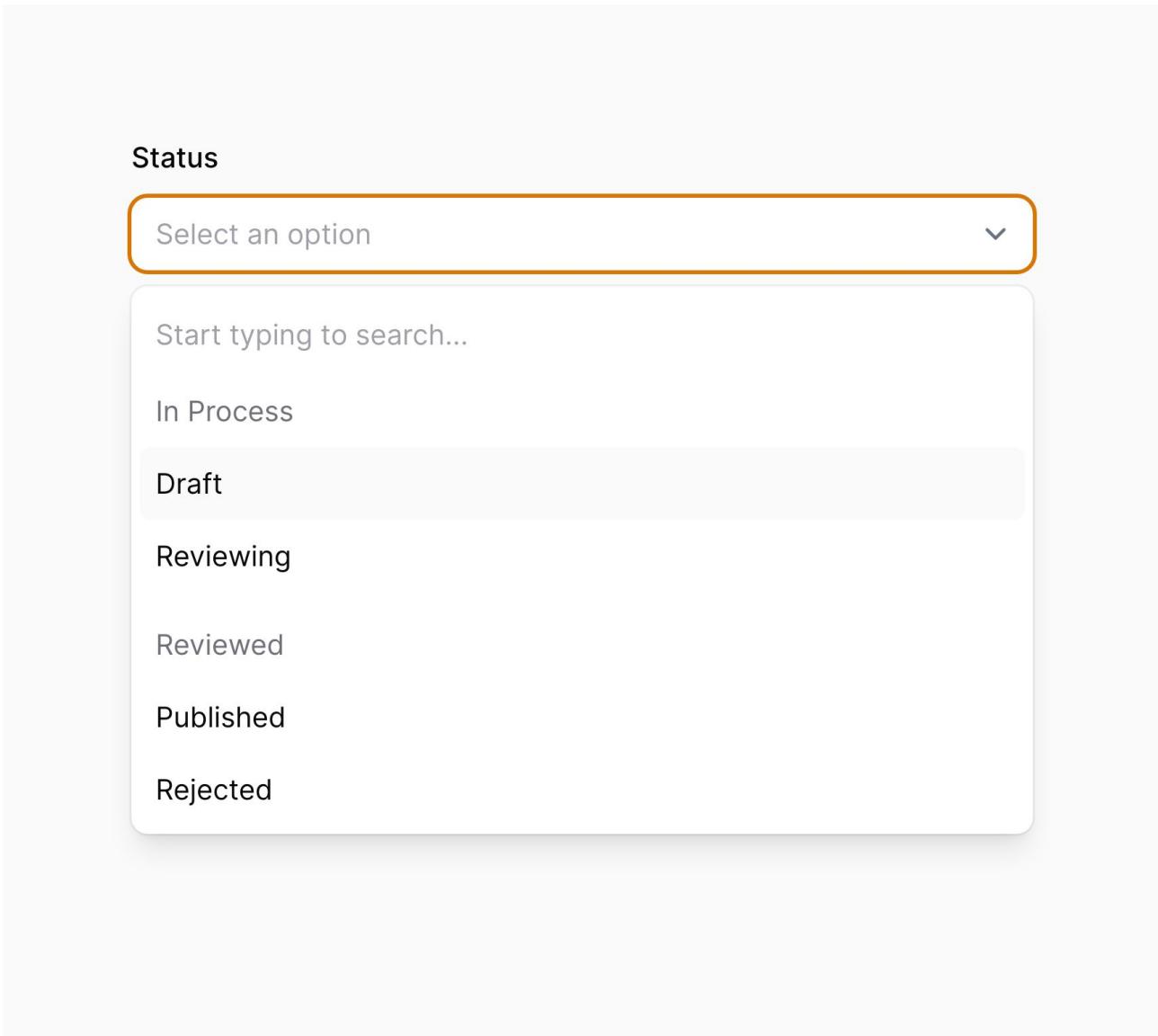
```
Select::make('technologies')
->multiple()
->searchable()
->getSearchResultsUsing(fn (string $search): array => Technology::where('name', 'like', "%{$search}%")->limit(50)->pluck('name', 'id')->toArray())
->getOptionLabelsUsing(fn (array $values): array => Technology::whereIn('id', $values)->pluck('name', 'id')->toArray()),
```

Grouping options

You can group options together under a label, to organize them better. To do this, you can pass an array of groups to `options()` or wherever you would normally pass an array of options. The keys of the array are used as group labels, and the values are arrays of options in that group:

```
use Filament\Forms\Components\Select;

Select::make('status')
->searchable()
->options([
    'In Process' => [
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
    ],
    'Reviewed' => [
        'published' => 'Published',
        'rejected' => 'Rejected',
    ],
])
```



Integrating with an Eloquent relationship

If you're building a form inside your Livewire component, make sure you have set up the [form's model](#). Otherwise, Filament doesn't know which model to use to retrieve the relationship from.

You may employ the `relationship()` method of the `Select` to configure a `BelongsTo` relationship to automatically retrieve options from. The `titleAttribute` is the name of a column that will be used to generate a label for each option:

```
use Filament\Forms\Components\Select;

Select::make('author_id')
->relationship(name: 'author', titleAttribute: 'name')
```

The `multiple()` method may be used in combination with `relationship()` to use a `BelongsToMany` relationship. Filament will load the options from the relationship, and save them back to the relationship's pivot table when the form is submitted. If a `name` is not provided, Filament will use the field name as the relationship name:

```
use Filament\Forms\Components>Select;

Select::make('technologies')
->multiple()
->relationship(titleAttribute: 'name')
```

When using `disabled()` with `multiple()` and `relationship()`, ensure that `disabled()` is called before `relationship()`. This ensures that the `dehydrated()` call from within `relationship()` is not overridden by the call from `disabled()`:

```
use Filament\Forms\Components>Select;

Select::make('technologies')
->multiple()
->disabled()
->relationship(titleAttribute: 'name')
```

Searching relationship options across multiple columns

By default, if the select is also searchable, Filament will return search results for the relationship based on the title column of the relationship. If you'd like to search across multiple columns, you can pass an array of columns to the `searchable()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
->relationship(name: 'author', titleAttribute: 'name')
->searchable(['name', 'email'])
```

Preloading relationship options

If you'd like to populate the searchable options from the database when the page is loaded, instead of when the user searches, you can use the `preload()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
->relationship(name: 'author', titleAttribute: 'name')
->searchable()
->preload()
```

Excluding the current record

When working with recursive relationships, you will likely want to remove the current record from the set of results.

This can be easily be done using the `ignoreRecord` argument:

```
use Filament\Forms\Components>Select;

Select::make('parent_id')
->relationship(name: 'parent', titleAttribute: 'name', ignoreRecord: true)
```

Customizing the relationship query

You may customize the database query that retrieves options using the third parameter of the `relationship()` method:

```
use Filament\Forms\Components>Select;
use Illuminate\Database\Eloquent\Builder;

Select::make('author_id')
    ->relationship(
        name: 'author',
        titleAttribute: 'name',
        modifyQueryUsing: fn (Builder $query) => $query->withTrashed(),
    )
```

If you would like to access the current search query in the `modifyQueryUsing()` function, you can inject `$search`.

Customizing the relationship option labels

If you'd like to customize the label of each option, maybe to be more descriptive, or to concatenate a first and last name, you could use a virtual column in your database migration:

```
$table->string('full_name')->virtualAs('concat(first_name, \' \', last_name)');
```

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'full_name')
```

Alternatively, you can use the `getOptionLabelFromRecordUsing()` method to transform an option's Eloquent model into a label:

```
use Filament\Forms\Components>Select;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

Select::make('author_id')
    ->relationship(
        name: 'author',
        modifyQueryUsing: fn (Builder $query) => $query->orderBy('first_name')-
    >orderBy('last_name'),
    )
    ->getOptionLabelFromRecordUsing(fn (Model $record) => "{$record->first_name} {$record-
    >last_name}")
    ->searchable(['first_name', 'last_name'])
```

Saving pivot data to the relationship

If you're using a `multiple()` relationship and your pivot table has additional columns, you can use the `pivotData()` method to specify the data that should be saved in them:

```
use Filament\Forms\Components>Select;

Select::make('primaryTechnologies')
    ->relationship(name: 'technologies', titleAttribute: 'name')
    ->multiple()
    ->pivotData([
        'is_primary' => true,
    ])
)
```

Creating a new option in a modal

You may define a custom form that can be used to create a new record and attach it to the `BelongsTo` relationship:

```
use Filament\Forms\Components>Select;

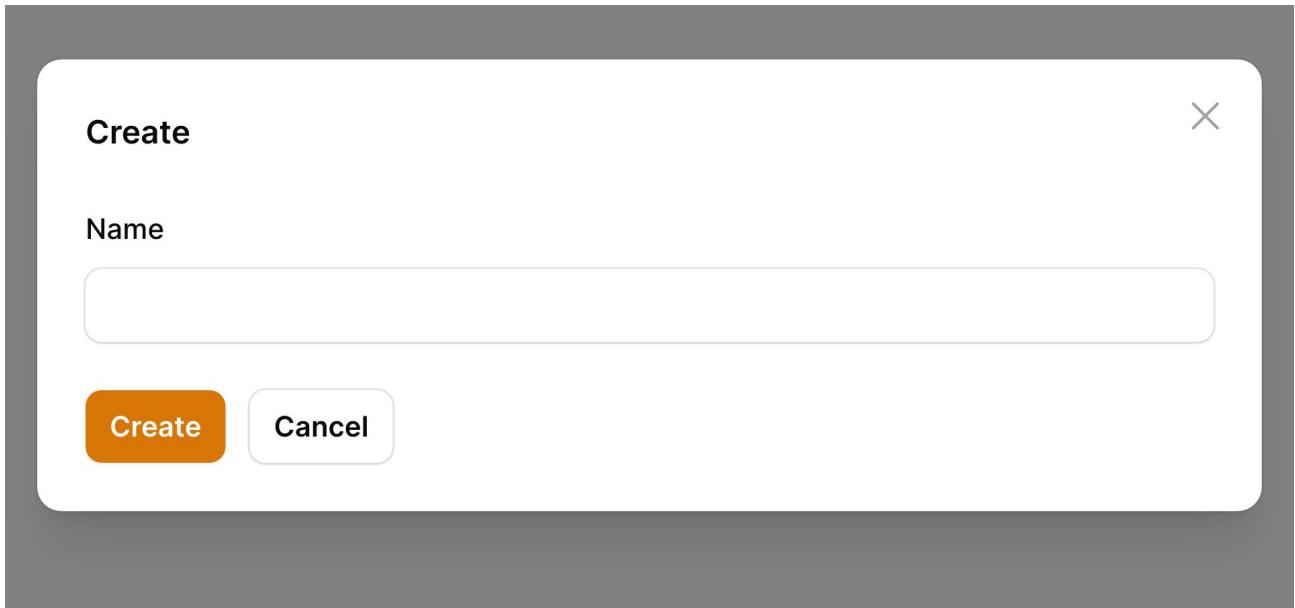
Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->createOptionForm([
        Forms\Components\TextInput::make('name')
            ->required(),
        Forms\Components\TextInput::make('email')
            ->required()
            ->email(),
    ]),
)
```

Author

Select an option



The form opens in a modal, where the user can fill it with data. Upon form submission, the new record is selected by the field.



Customizing new option creation

You can customize the creation process of the new option defined in the form using the `createOptionUsing()` method, which should return the primary key of the newly created record:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->createOptionForm([
        // ...
    ])
    ->createOptionUsing(function (array $data): int {
        return auth()->user()->team->members()->create($data)->getKey();
}),

```

Editing the selected option in a modal

You may define a custom form that can be used to edit the selected record and save it back to the `BelongsTo` relationship:

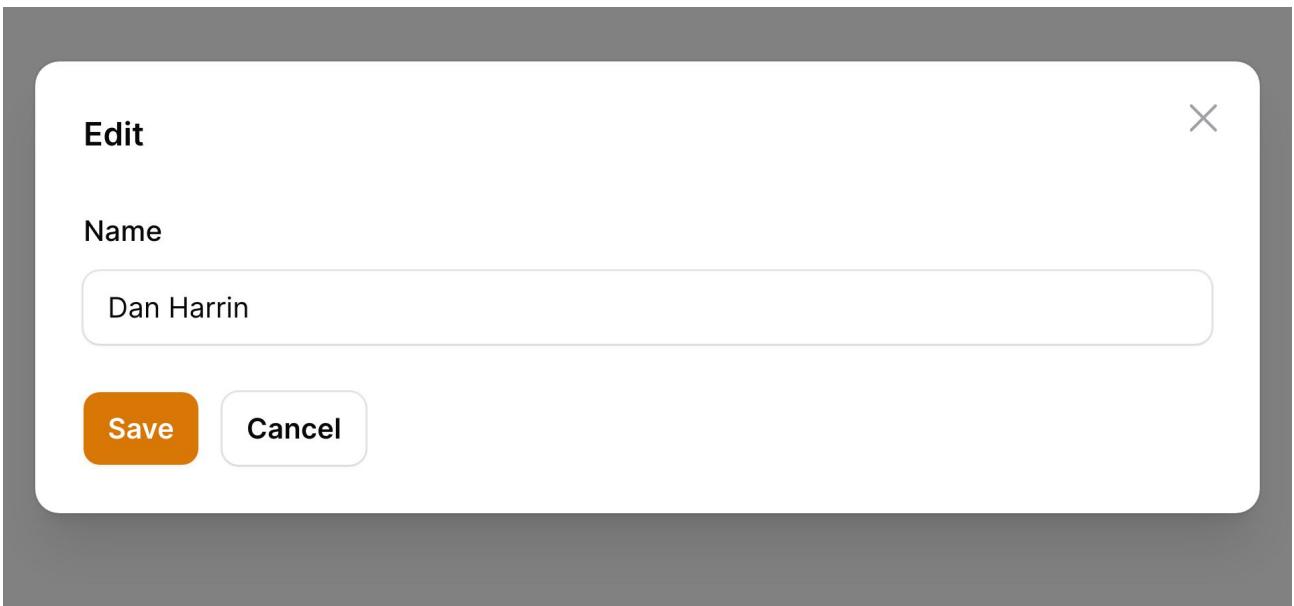
```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->editOptionForm([
        Forms\Components\TextInput::make('name')
            ->required(),
        Forms\Components\TextInput::make('email')
            ->required()
            ->email(),
    ]),

```

A screenshot of a modal window titled "Author". Inside the modal, there is a single input field containing the text "Dan Harrin". To the right of the input field is a small edit icon (pencil symbol) and a dropdown arrow icon.

The form opens in a modal, where the user can fill it with data. Upon form submission, the data from the form is saved back to the record.



Handling `MorphTo` relationships

`MorphTo` relationships are special, since they give the user the ability to select records from a range of different models. Because of this, we have a dedicated `MorphToSelect` component which is not actually a select field, rather 2 select fields inside afieldset. The first select field allows you to select the type, and the second allows you to select the record of that type.

To use the `MorphToSelect`, you must pass `types()` into the component, which tell it how to render options for different types:

```
use Filament\Forms\Components\MorphToSelect;

MorphToSelect::make('commentable')
->types([
    MorphToSelect\Type::make(Product::class)
        ->titleAttribute('name'),
    MorphToSelect\Type::make(Post::class)
        ->titleAttribute('title'),
])
])
```

Customizing the option labels for each morphed type

The `titleAttribute()` is used to extract the titles out of each product or post. If you'd like to customize the label of each option, you can use the `getOptionLabelFromRecordUsing()` method to transform the Eloquent model into a label:

```
use Filament\Forms\Components\MorphToSelect;

MorphToSelect::make('commentable')
->types([
    MorphToSelect\Type::make(Product::class)
        ->getOptionLabelFromRecordUsing(fn (Product $record): string => "{$record->name} - {$record->slug}"),
    MorphToSelect\Type::make(Post::class)
        ->titleAttribute('title'),
])
])
```

Customizing the relationship query for each morphed type

You may customize the database query that retrieves options using the `modifyOptionsQueryUsing()` method:

```
use Filament\Forms\Components\MorphToSelect;
use Illuminate\Database\Eloquent\Builder;

MorphToSelect::make('commentable')
->types([
    MorphToSelect\Type::make(Product::class)
        ->titleAttribute('name')
        ->modifyOptionsQueryUsing(fn (Builder $query) => $query->whereBelongsTo($this->team)),
    MorphToSelect\Type::make(Post::class)
        ->titleAttribute('title')
        ->modifyOptionsQueryUsing(fn (Builder $query) => $query->whereBelongsTo($this->team)),
])
])
```

Many of the same options in the select field are available for `MorphToSelect`, including `searchable()`, `preload()`, `native()`, `allowHtml()`, and `optionsLimit()`.

Allowing HTML in the option labels

By default, Filament will escape any HTML in the option labels. If you'd like to allow HTML, you can use the `allowHtml()` method:

```
use Filament\Forms\Components>Select;

Select::make('technology')
->options([
    'tailwind' => '<span class="text-blue-500">Tailwind</span>',
    'alpine' => '<span class="text-green-500">Alpine</span>',
    'laravel' => '<span class="text-red-500">Laravel</span>',
    'livewire' => '<span class="text-pink-500">Livewire</span>',
])
->searchable()
->allowHtml()
```

Be aware that you will need to ensure that the HTML is safe to render, otherwise your application will be vulnerable to XSS attacks.

Disable placeholder selection

You can prevent the placeholder (null option) from being selected using the `selectablePlaceholder()` method:

```
use Filament\Forms\Components>Select;

Select::make('status')
->options([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
->default('draft')
->selectablePlaceholder(false)
```

Disabling specific options

You can disable specific options using the `disableOptionWhen()` method. It accepts a closure, in which you can check if the option with a specific `$value` should be disabled:

```
use Filament\Forms\Components>Select;

Select::make('status')
->options([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
->default('draft')
->disableOptionWhen(fn (string $value): bool => $value === 'published')
```

If you want to retrieve the options that have not been disabled, e.g. for validation purposes, you can do so using `getEnabledOptions()`:

```
use Filament\Forms\Components>Select;

Select::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->default('draft')
    ->disableOptionWhen(fn (string $value): bool => $value === 'published')
    ->in(fn (Select $component): array => array_keys($component->getEnabledOptions()))

```

Adding affix text aside the field

You may place text before and after the input using the `prefix()` and `suffix()` methods:

```
use Filament\Forms\Components>Select;

Select::make('domain')
    ->prefix('https://')
    ->suffix('.com')
```

Domain

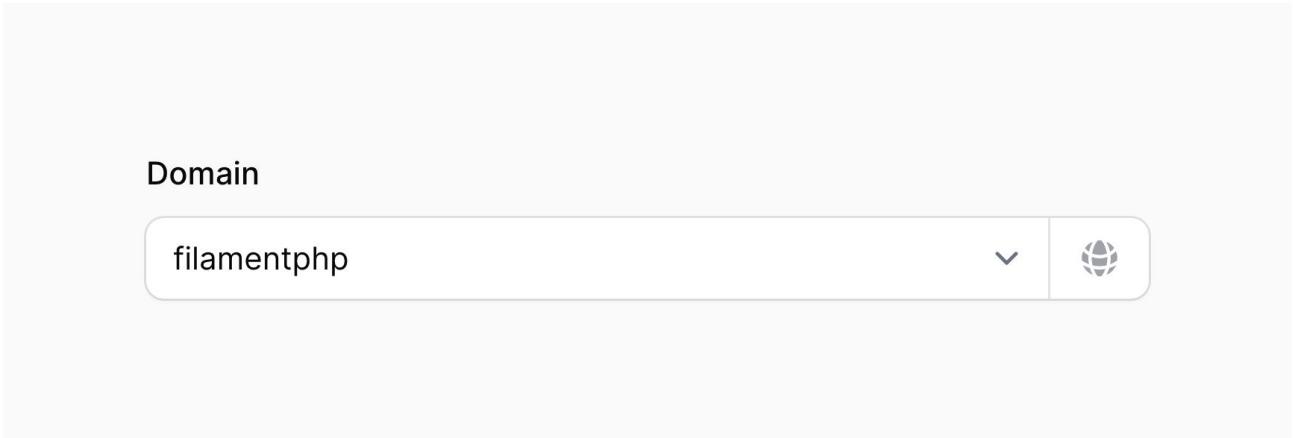


Using icons as affixes

You may place an `icon` before and after the input using the `prefixIcon()` and `suffixIcon()` methods:

```
use Filament\Forms\Components>Select;

Select::make('domain')
    ->suffixIcon('heroicon-m-globe-alt')
```



Setting the affix icon's color

Affix icons are gray by default, but you may set a different color using the `prefixIconColor()` and `suffixIconColor()` methods:

```
use Filament\Forms\Components>Select;

Select::make('domain')
    ->suffixIcon('heroicon-m-check-circle')
    ->suffixIconColor('success')
```

Setting a custom loading message

When you're using a searchable select or multi-select, you may want to display a custom message while the options are loading. You can do this using the `loadingMessage()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable()
    ->loadingMessage('Loading authors...')
```

Setting a custom no search results message

When you're using a searchable select or multi-select, you may want to display a custom message when no search results are found. You can do this using the `noSearchResultsMessage()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable()
    ->noSearchResultsMessage('No authors found.')
```

Setting a custom search prompt

When you're using a searchable select or multi-select, you may want to display a custom message when the user has not yet entered a search term. You can do this using the `searchPrompt()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable(['name', 'email'])
    ->searchPrompt('Search authors by their name or email address')
```

Setting a custom searching message

When you're using a searchable select or multi-select, you may want to display a custom message while the search results are being loaded. You can do this using the `searchingMessage()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable()
    ->searchingMessage('Searching authors...')
```

Tweaking the search debounce

By default, Filament will wait 1000 milliseconds (1 second) before searching for options when the user types in a searchable select or multi-select. It will also wait 1000 milliseconds between searches, if the user is continuously typing into the search input. You can change this using the `searchDebounce()` method:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable()
    ->searchDebounce(500)
```

Ensure that you are not lowering the debounce too much, as this may cause the select to become slow and unresponsive due to a high number of network requests to retrieve options from server.

Limiting the number of options

You can limit the number of options that are displayed in a searchable select or multi-select using the `optionsLimit()` method. The default is 50:

```
use Filament\Forms\Components>Select;

Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->searchable()
    ->optionsLimit(20)
```

Ensure that you are not raising the limit too high, as this may cause the select to become slow and unresponsive due to high in-browser memory usage.

Select validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to selects.

Selected items validation

You can validate the minimum and maximum number of items that you can select in a [multi-select](#) by setting the `minItems()` and `maxItems()` methods:

```
use Filament\Forms\Components\Select;

Select::make('technologies')
    ->multiple()
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
    ->minItems(1)
    ->maxItems(3)
```

Customizing the select action objects

This field uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to [customize it](#) or [customize its modal](#). The following methods are available to customize the actions:

- `createOptionAction()`
- `editOptionAction()`
- `manageOptionActions()` (for customizing both the create and edit option actions at once)

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components>Select;

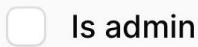
Select::make('author_id')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->createOptionAction(
        fn (Action $action) => $action->modalWidth('3xl'),
    )
```

Checkbox

Overview

The checkbox component, similar to a [toggle](#), allows you to interact a boolean value.

```
use Filament\Forms\Components\Checkbox;  
  
Checkbox::make('is_admin')
```



If you're saving the boolean value using Eloquent, you should be sure to add a `boolean` [cast](#) to the model property:

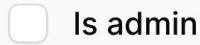
```
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model  
{  
    protected $casts = [  
        'is_admin' => 'boolean',  
    ];  
  
    // ...  
}
```

Positioning the label above

Checkbox fields have two layout modes, inline and stacked. By default, they are inline.

When the checkbox is inline, its label is adjacent to it:

```
use Filament\Forms\Components\Checkbox;  
  
Checkbox::make('is_admin')->inline()
```



When the checkbox is stacked, its label is above it:

```
use Filament\Forms\Components\Checkbox;  
  
Checkbox::make('is_admin')->inline(false)
```

Is admin



Checkbox validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to checkboxes.

Accepted validation

You may ensure that the checkbox is checked using the `accepted()` method:

```
use Filament\Forms\Components\Checkbox;  
  
Checkbox::make('terms_of_service')  
->accepted()
```

Declined validation

You may ensure that the checkbox is not checked using the `declined()` method:

```
use Filament\Forms\Components\Checkbox;  
  
Checkbox::make('is_under_18')  
->declined()
```

Toggle

Overview

The toggle component, similar to a [checkbox](#), allows you to interact with a boolean value.

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
```



If you're saving the boolean value using Eloquent, you should be sure to add a [boolean cast](#) to the model property:

```
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $casts = [
        'is_admin' => 'boolean',
    ];

    // ...
}
```

Adding icons to the toggle button

Toggles may also use an [icon](#) to represent the "on" and "off" state of the button. To add an icon to the "on" state, use the [onIcon\(\)](#) method. To add an icon to the "off" state, use the [offIcon\(\)](#) method:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
    ->onIcon('heroicon-m-bolt')
    ->offIcon('heroicon-m-user')
```



Is admin

Customizing the color of the toggle button

You may also customize the color representing the "on" or "off" state of the toggle. These may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`. To add a color to the "on" state, use the `onColor()` method. To add a color to the "off" state, use the `offColor()` method:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
    ->onColor('success')
    ->offColor('danger')
```



Is admin



Is admin

Positioning the label above

Toggle fields have two layout modes, inline and stacked. By default, they are inline.

When the toggle is inline, its label is adjacent to it:

```
use Filament\Forms\Components\Toggle;

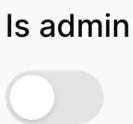
Toggle::make('is_admin')
->inline()
```



When the toggle is stacked, its label is above it:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
->inline(false)
```



Toggle validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to toggles.

Accepted validation

You may ensure that the toggle is "on" using the `accepted()` method:

```
use Filament\Forms\Components\Toggle;

Toggle::make('terms_of_service')
->accepted()
```

Declined validation

You may ensure that the toggle is "off" using the `declined()` method:

```
use Filament\Forms\Components\Toggle;  
  
Toggle::make('is_under_18')  
->declined()
```

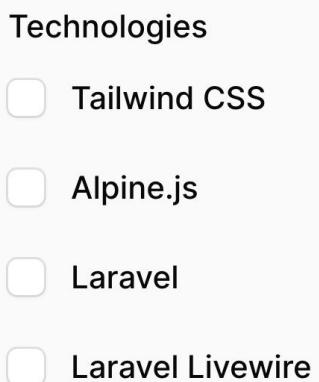
Checkbox List

Overview

The checkbox list component allows you to select multiple values from a list of predefined options:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
)
```



These options are returned in JSON format. If you're saving them using Eloquent, you should be sure to add an [array cast](#) to the model property:

```
use Illuminate\Database\Eloquent\Model;

class App extends Model
{
    protected $casts = [
        'technologies' => 'array',
    ];
    // ...
}
```

Setting option descriptions

You can optionally provide descriptions to each option using the `descriptions()` method. This method accepts an array of plain text strings, or instances of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even markdown, in the descriptions:

```
use Filament\Forms\Components\CheckboxList;
use Illuminate\Support\HtmlString;

CheckboxList::make('technologies')
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
    ->descriptions([
        'tailwind' => 'A utility-first CSS framework for rapidly building modern websites without ever leaving your HTML.',
        'alpine' => new HtmlString('A rugged, minimal tool for composing behavior directly in your markup.'),  

        'laravel' => str('A **web application** framework with expressive, elegant syntax.')->inlineMarkdown()->toHtmlString(),
        'livewire' => 'A full-stack framework for Laravel building dynamic interfaces simple, without leaving the comfort of Laravel.',
    ])
)
```

Technologies

Tailwind CSS

A utility-first CSS framework for rapidly building modern websites without ever leaving your HTML.

Alpine.js

A rugged, minimal tool for composing behavior **directly in your markup.**

Laravel

A **web application** framework with expressive, elegant syntax.

Laravel Livewire

A full-stack framework for Laravel building dynamic interfaces simple, without leaving the comfort of Laravel.

Be sure to use the same `key` in the descriptions array as the `key` in the option array so the right description matches the right option.

Splitting options into columns

You may split options into columns by using the `columns()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->columns(2)
```

Technologies

Tailwind CSS

Laravel

Alpine.js

Laravel Livewire

This method accepts the same options as the `columns()` method of the `grid`. This allows you to responsively customize the number of columns at various breakpoints.

Setting the grid direction

By default, when you arrange checkboxes into columns, they will be listed in order vertically. If you'd like to list them horizontally, you may use the `gridDirection('row')` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->columns(2)
    ->gridDirection('row')
```

Technologies

Tailwind CSS

Alpine.js

Laravel

Laravel Livewire

Disabling specific options

You can disable specific options using the `disableOptionWhen()` method. It accepts a closure, in which you can check if the option with a specific `$value` should be disabled:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
    ->disableOptionWhen(fn (string $value): bool => $value === 'livewire')
```

If you want to retrieve the options that have not been disabled, e.g. for validation purposes, you can do so using `getEnabledOptions()`:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
        'heroicons' => 'SVG icons',
    ])
    ->disableOptionWhen(fn (string $value): bool => $value === 'heroicons')
    ->in(fn (CheckboxList $component): array => array_keys($component->getEnabledOptions())))
```

Searching options

You may enable a search input to allow easier access to many options, using the `searchable()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->searchable()
```

Technologies

 Start typing to search...

Tailwind CSS

Alpine.js

Laravel

Laravel Livewire

Bulk toggling checkboxes

You may allow users to toggle all checkboxes at once using the `bulkToggleable()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->bulkToggleable()
```

Technologies

Select all

Tailwind CSS

Alpine.js

Laravel

Laravel Livewire

Integrating with an Eloquent relationship

If you're building a form inside your Livewire component, make sure you have set up the [form's model](#). Otherwise, Filament doesn't know which model to use to retrieve the relationship from.

You may employ the `relationship()` method of the `CheckboxList` to point to a `BelongsToMany` relationship. Filament will load the options from the relationship, and save them back to the relationship's pivot table when the form is submitted. The `titleAttribute` is the name of a column that will be used to generate a label for each option:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->relationship(titleAttribute: 'name')
```

When using `disabled()` with `relationship()`, ensure that `disabled()` is called before `relationship()`. This ensures that the `dehydrated()` call from within `relationship()` is not overridden by the call from `disabled()`:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->disabled()
    ->relationship(titleAttribute: 'name')
```

Customizing the relationship query

You may customize the database query that retrieves options using the `modifyOptionsQueryUsing` parameter of the `relationship()` method:

```
use Filament\Forms\Components\CheckboxList;
use Illuminate\Database\Eloquent\Builder;

CheckboxList::make('technologies')
    ->relationship(
        titleAttribute: 'name',
        modifyQueryUsing: fn (Builder $query) => $query->withTrashed(),
    )
)
```

Customizing the relationship option labels

If you'd like to customize the label of each option, maybe to be more descriptive, or to concatenate a first and last name, you could use a virtual column in your database migration:

```
$table->string('full_name')->virtualAs('concat(first_name, \' \', last_name)');
```

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('authors')
    ->relationship(titleAttribute: 'full_name')
```

Alternatively, you can use the `getOptionLabelFromRecordUsing()` method to transform an option's Eloquent model into a label:

```
use Filament\Forms\Components\CheckboxList;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

CheckboxList::make('authors')
    ->relationship(
        modifyQueryUsing: fn (Builder $query) => $query->orderBy('first_name')-
    >orderBy('last_name'),
    )
    ->getOptionLabelFromRecordUsing(fn (Model $record) => "{$record->first_name} {$record-
    >last_name}")
```

Saving pivot data to the relationship

If your pivot table has additional columns, you can use the `pivotData()` method to specify the data that should be saved in them:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('primaryTechnologies')
    ->relationship(name: 'technologies', titleAttribute: 'name')
    ->pivotData([
        'is_primary' => true,
    ])
```

Setting a custom no search results message

When you're using a searchable checkbox list, you may want to display a custom message when no search results are found. You can do this using the `noSearchResultsMessage()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->searchable()
    ->noSearchResultsMessage('No technologies found.')
```

Setting a custom search prompt

When you're using a searchable checkbox list, you may want to tweak the search input's placeholder when the user has not yet entered a search term. You can do this using the `searchPrompt()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->searchable()
    ->searchPrompt('Search for a technology')
```

Tweaking the search debounce

By default, Filament will wait 1000 milliseconds (1 second) before searching for options when the user types in a searchable checkbox list. It will also wait 1000 milliseconds between searches if the user is continuously typing into the search input. You can change this using the `searchDebounce()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->searchable()
    ->searchDebounce(500)
```

Customizing the checkbox list action objects

This field uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to [customize it](#). The following methods are available to customize the actions:

- `selectAllAction()`
- `deselectAllAction()`

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->options([
        // ...
    ])
    ->selectAllAction(
        fn (Action $action) => $action->label('Select all technologies'),
    )
```

Radio

Overview

The radio input provides a radio button group for selecting a single value from a list of predefined options:

```
use Filament\Forms\Components\Radio;

Radio::make('status')
->options([
    'draft' => 'Draft',
    'scheduled' => 'Scheduled',
    'published' => 'Published'
])
```

Status

- Draft
- Scheduled
- Published

Setting option descriptions

You can optionally provide descriptions to each option using the `descriptions()` method:

```
use Filament\Forms\Components\Radio;

Radio::make('status')
->options([
    'draft' => 'Draft',
    'scheduled' => 'Scheduled',
    'published' => 'Published'
])
->descriptions([
    'draft' => 'Is not visible.',
    'scheduled' => 'Will be visible.',
    'published' => 'Is visible.'
])
```

Status

Draft

Is not visible.

Scheduled

Will be visible.

Published

Is visible.

Be sure to use the same `key` in the descriptions array as the `key` in the option array so the right description matches the right option.

Boolean options

If you want a simple boolean radio button group, with "Yes" and "No" options, you can use the `boolean()` method:

```
Radio::make('feedback')
->label('Like this post?')
->boolean()
```

Like this post?

Yes

No

Positioning the options inline with the label

You may wish to display the options `inline()` with the label instead of below it:

```
Radio::make('feedback')
->label('Like this post?')
->boolean()
->inline()
```

Like this post?

Yes No

Positioning the options inline with each other but below the label

You may wish to display the options `inline()` with each other but below the label:

```
Radio::make('feedback')
->label('Like this post?')
->boolean()
->inline()
->inlineLabel(false)
```

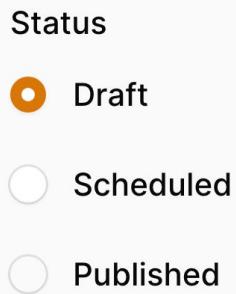


Disabling specific options

You can disable specific options using the `disableOptionWhen()` method. It accepts a closure, in which you can check if the option with a specific `$value` should be disabled:

```
use Filament\Forms\Components\Radio;

Radio::make('status')
->options([
    'draft' => 'Draft',
    'scheduled' => 'Scheduled',
    'published' => 'Published',
])
->disableOptionWhen(fn (string $value): bool => $value === 'published')
```



If you want to retrieve the options that have not been disabled, e.g. for validation purposes, you can do so using `getEnabledOptions()`:

```
use Filament\Forms\Components\Radio;

Radio::make('status')
->options([
    'draft' => 'Draft',
    'scheduled' => 'Scheduled',
    'published' => 'Published',
])
->disableOptionWhen(fn (string $value): bool => $value === 'published')
->in(fn (Radio $component): array => array_keys($component->getEnabledOptions()))
```

Date Time Picker

Overview

The date-time picker provides an interactive interface for selecting a date and/or a time.

```
use Filament\Forms\Components\DatePicker;
use Filament\Forms\Components\DateTimePicker;
use Filament\Forms\Components\TimePicker;

DateTimePicker::make('published_at')
DatePicker::make('date_of_birth')
TimePicker::make('alarm_at')
```

Published at

dd/mm/yyyy, --:--:--
□

Date of birth

dd/mm/yyyy
□

Alarm at

--:--:--
⌚

Customizing the storage format

You may customize the format of the field when it is saved in your database, using the `format()` method. This accepts a string date format, using [PHP date formatting tokens](#):

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
->format('d/m/Y')
```

Disabling the seconds input

When using the time picker, you may disable the seconds input using the `seconds(false)` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->seconds(false)
```

Published at

dd/mm/yyyy, --:--



Timezones

If you'd like users to be able to manage dates in their own timezone, you can use the `timezone()` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->timezone('America/New_York')
```

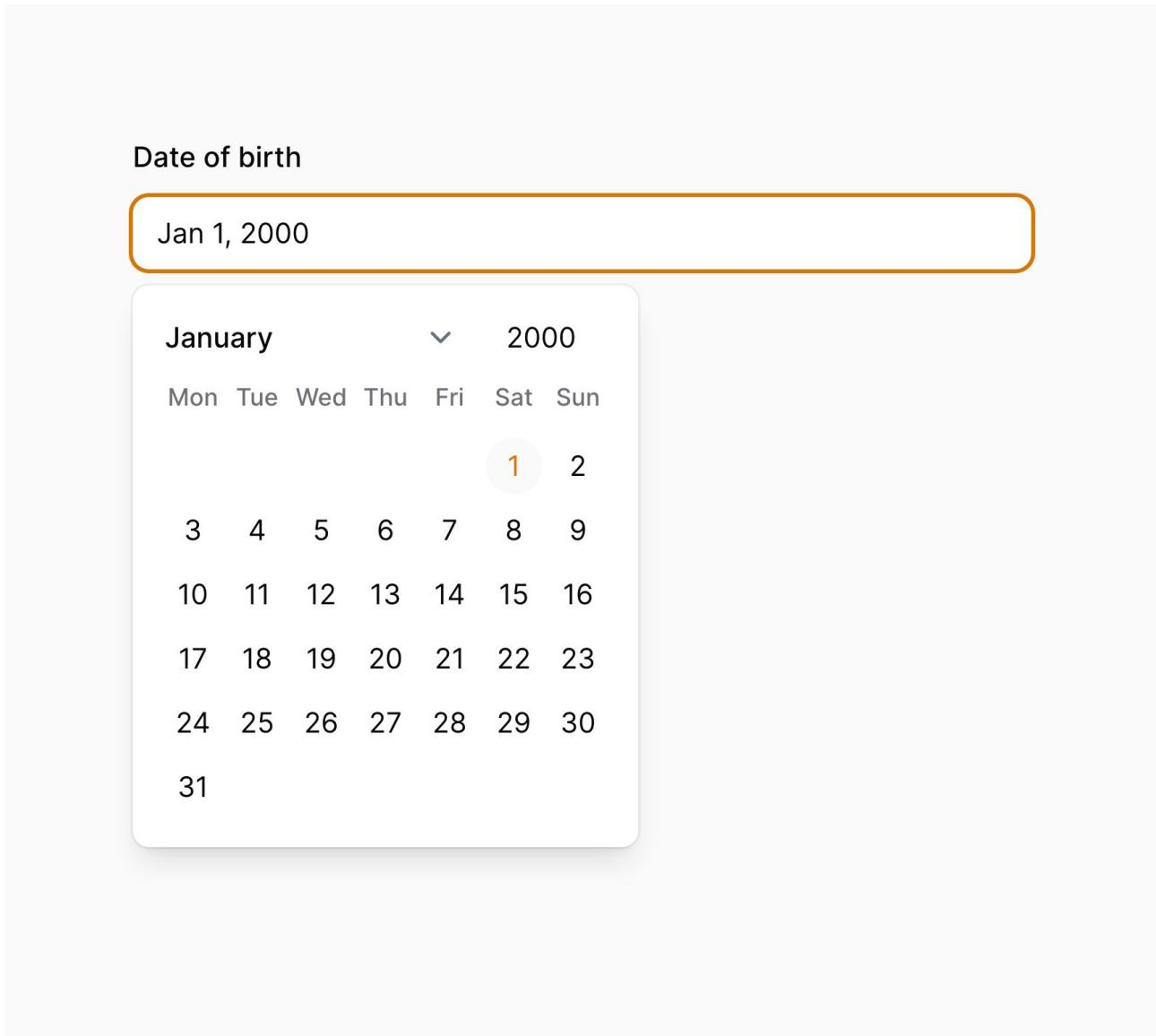
While dates will still be stored using the app's configured timezone, the date will now load in the new timezone, and it will be converted back when the form is saved.

Enabling the JavaScript date picker

By default, Filament uses the native HTML5 date picker. You may enable a more customizable JavaScript date picker using the `native(false)` method:

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
    ->native(false)
```



Please be aware that while being accessible, the JavaScript date picker does not support full keyboard input in the same way that the native date picker does. If you require full keyboard input, you should use the native date picker.

Customizing the display format

You may customize the display format of the field, separately from the format used when it is saved in your database. For this, use the `displayFormat()` method, which also accepts a string date format, using [PHP date formatting tokens](#):

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
    ->native(false)
    ->displayFormat('d/m/Y')
```

Date of birth

01/01/2000

You may also configure the locale that is used when rendering the display, if you want to use different locale from your app config. For this, you can use the `locale()` method:

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
->native(false)
->displayFormat('d F Y')
->locale('fr')
```

Configuring the time input intervals

You may customize the input interval for increasing/decreasing the hours/minutes /seconds using the `hoursStep()`, `minutesStep()` or `secondsStep()` methods:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
->native(false)
->hoursStep(2)
->minutesStep(15)
->secondsStep(10)
```

Configuring the first day of the week

In some countries, the first day of the week is not Monday. To customize the first day of the week in the date picker, use the `firstDayOfWeek()` method on the component. 0 to 7 are accepted values, with Monday as 1 and Sunday as 7 or 0:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
->native(false)
->firstDayOfWeek(7)
```

Published at

Jan 1, 2000



There are additionally convenient helper methods to set the first day of the week more semantically:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->native(false)
    ->weekStartsOnMonday()

DateTimePicker::make('published_at')
    ->native(false)
    ->weekStartsOnSunday()
```

Disabling specific dates

To prevent specific dates from being selected:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('date')
->native(false)
->disabledDates(['2000-01-03', '2000-01-15', '2000-01-20'])
```

Date

Jan 1, 2000



Closing the picker when a date is selected

To close the picker when a date is selected, you can use the `closeOnDateSelection()` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('date')
->native(false)
->closeOnDateSelection()
```

Autocompleting dates with a datalist

Unless you're using the [JavaScript date picker](#), you may specify `datalist` options for a date picker using the `datalist()` method:

```
use Filament\Forms\Components\TimePicker;

TimePicker::make('appointment_at')
    ->datalist([
        '09:00',
        '09:30',
        '10:00',
        '10:30',
        '11:00',
        '11:30',
        '12:00',
    ])
)
```

Datalists provide autocomplete options to users when they use the picker. However, these are purely recommendations, and the user is still able to type any value into the input. If you're looking to strictly limit users to a set of predefined options, check out the [select field](#).

Adding affix text aside the field

You may place text before and after the input using the `prefix()` and `suffix()` methods:

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date')
    ->prefix('Starts')
    ->suffix('at midnight')
```

Date

Starts

01/01/2000



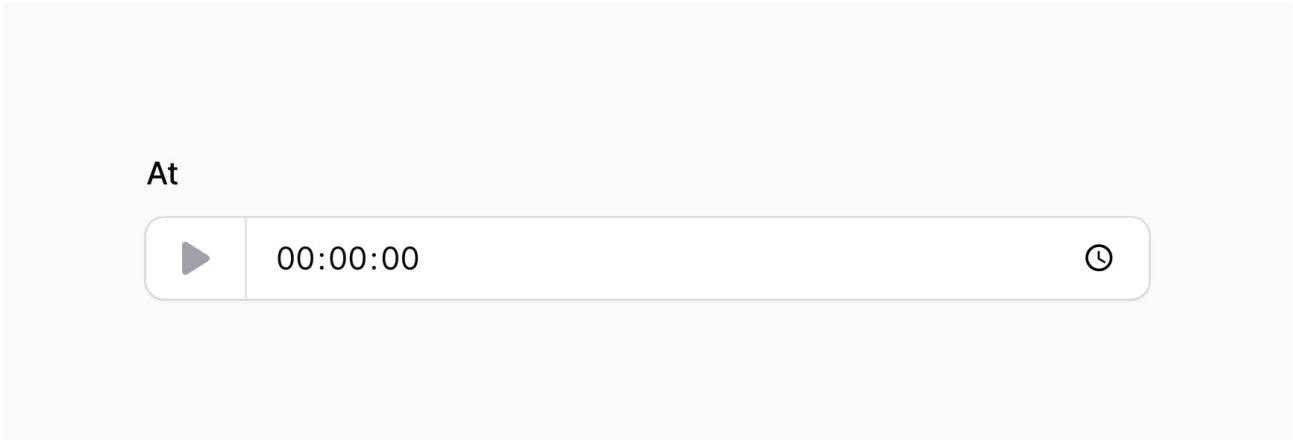
at midnight

Using icons as affixes

You may place an [icon](#) before and after the input using the `prefixIcon()` and `suffixIcon()` methods:

```
use Filament\Forms\Components\TimePicker;

TimePicker::make('at')
    ->prefixIcon('heroicon-m-play')
```



Setting the affix icon's color

Affix icons are gray by default, but you may set a different color using the `prefixIconColor()` and `suffixIconColor()` methods:

```
use Filament\Forms\Components\TimePicker;

TimePicker::make('at')
    ->prefixIcon('heroicon-m-check-circle')
    ->prefixIconColor('success')
```

Making the field read-only

Not to be confused with disabling the field, you may make the field "read-only" using the `readonly()` method:

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
    ->readonly()
```

Please note that this setting is only enforced on native date pickers. If you're using the JavaScript date picker, you'll need to use `disabled()`.

There are a few differences, compared to `disabled()`:

- When using `readonly()`, the field will still be sent to the server when the form is submitted. It can be mutated with the browser console, or via JavaScript. You can use `dehydrated(false)` to prevent this.
- There are no styling changes, such as less opacity, when using `readonly()`.
- The field is still focusable when using `readonly()`.

Date-time picker validation

As well as all rules listed on the validation page, there are additional rules that are specific to date-time pickers.

Max date / min date validation

You may restrict the minimum and maximum date that can be selected with the picker. The `minDate()` and `maxDate()` methods accept a `DateTime` instance (e.g. `Carbon`), or a string:

```
use Filament\Forms\Components\DatePicker;

DatePicker::make('date_of_birth')
->native(false)
->minDate(now() ->subYears(150))
->maxDate(now())
```

File Upload

Overview

The file upload field is based on [Filepond](#).

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
```

Attachment

Drag & Drop your files or [Browse](#)

Filament also supports [spatie/laravel-medialibrary](#). See our [plugin documentation](#) for more information.

Configuring the storage disk and directory

By default, files will be uploaded publicly to your storage disk defined in the [configuration file](#). You can also set the `FILAMENT_FILESYSTEM_DISK` environment variable to change this.

To correctly preview images and other files, FilePond requires files to be served from the same domain as the app, or the appropriate CORS headers need to be present. Ensure that the `APP_URL` environment variable is correct, or modify the `filesystem` driver to set the correct URL. If you're hosting files on a separate domain like S3, ensure that CORS headers are set up.

To change the disk and directory for a specific field, and the visibility of files, use the `disk()`, `directory()` and `visibility()` methods:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->disk('s3')
->directory('form-attachments')
->visibility('private')
```

It is the responsibility of the developer to delete these files from the disk if they are removed, as Filament is unaware if they are depended on elsewhere. One way to do this automatically is observing a [model event](#).

Uploading multiple files

You may also upload multiple files. This stores URLs in JSON:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
```

If you're saving the file URLs using Eloquent, you should be sure to add an `array` [cast](#) to the model property:

```
use Illuminate\Database\Eloquent\Model;

class Message extends Model
{
    protected $casts = [
        'attachments' => 'array',
    ];

    // ...
}
```

Controlling file names

By default, a random file name will be generated for newly-uploaded files. This is to ensure that there are never any conflicts with existing files.

Security implications of controlling file names

Before using the `preserveFilenames()` or `getUploadedFileNameForStorageUsing()` methods, please be aware of the security implications. If you allow users to upload files with their own file names, there are ways that they can exploit this to upload malicious files. **This applies even if you use the `acceptedFileTypes()` method** to restrict the types of files that can be uploaded, since it uses Laravel's `mimetypes` rule which does not validate the extension of the file, only its mime type, which could be manipulated.

This is specifically an issue with the `getClientOriginalName()` method on the `TemporaryUploadedFile` object, which the `preserveFilenames()` method uses. By default, Livewire generates a random file name for each file uploaded, and uses the mime type of the file to determine the file extension.

Using these methods **with the `local` or `public` filesystem disks** will make your app vulnerable to remote code execution if the attacker uploads a PHP file with a deceptive mime type. **Using an S3 disk protects you from this specific attack vector**, as S3 will not execute PHP files in the same way that your server might when serving files from local storage.

If you are using the `local` or `public` disk, you should consider using the `storeFileNamesIn()` method to store the original file names in a separate column in your database, and keep the randomly generated file names in the file system. This way, you can still display the original file names to users, while keeping the file system secure.

On top of this security issue, you should also be aware that allowing users to upload files with their own file names can lead to conflicts with existing files, and can make it difficult to manage your storage. Users could upload files with the same name and overwrite the other's content if you do not scope them to a specific directory, so these features should in all cases only be accessible to trusted users.

Preserving original file names

Important: Before using this feature, please ensure that you have read the [security implications](#).

To preserve the original filenames of the uploaded files, use the `preserveFilenames()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->preserveFilenames()
```

Generating custom file names

Important: Before using this feature, please ensure that you have read the [security implications](#).

You may completely customize how file names are generated using the `getUploadedFileNameForStorageUsing()` method, and returning a string from the closure based on the `$file` that was uploaded:

```
use Livewire\Features\SupportFileUploads\TemporaryUploadedFile;

FileUpload::make('attachment')
->getUploadedFileNameForStorageUsing(
    fn (TemporaryUploadedFile $file): string => (string) str($file->getClientOriginalName())
        ->prepend('custom-prefix-'),
)
```

Storing original file names independently

You can keep the randomly generated file names, while still storing the original file name, using the `storeFileNamesIn()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->storeFileNamesIn('attachment_file_names')
```

`attachment_file_names` will now store the original file names of your uploaded files, so you can save them to the database when the form is submitted. If you're uploading `multiple()` files, make sure that you add an `array cast` to this Eloquent model property too.

Avatar mode

You can enable avatar mode for your file upload field using the `avatar()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('avatar')
->avatar()
```

This will only allow images to be uploaded, and when they are, it will display them in a compact circle layout that is perfect for avatars.

This feature pairs well with the [circle cropper](#).

Image editor

You can enable an image editor for your file upload field using the `imageEditor()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
    ->image()
    ->imageEditor()
```

You can open the editor once you upload an image by clicking the pencil icon. You can also open the editor by clicking the pencil icon on an existing image, which will remove and re-upload it on save.

Allowing users to crop images to aspect ratios

You can allow users to crop images to a set of specific aspect ratios using the `imageEditorAspectRatios()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
    ->image()
    ->imageEditor()
    ->imageEditorAspectRatios([
        '16:9',
        '4:3',
        '1:1',
    ])
```

You can also allow users to choose no aspect ratio, "free cropping", by passing `null` as an option:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
    ->image()
    ->imageEditor()
    ->imageEditorAspectRatios([
        null,
        '16:9',
        '4:3',
        '1:1',
    ])
```

Setting the image editor's mode

You can change the mode of the image editor using the `imageEditorMode()` method, which accepts either `1`, `2` or `3`. These options are explained in the [Copper.js documentation](#):

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
    ->image()
    ->imageEditor()
    ->imageEditorMode(2)
```

Customizing the image editor's empty fill color

By default, the image editor will make the empty space around the image transparent. You can customize this using the `imageEditorEmptyFillColor()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
->image()
->imageEditor()
->imageEditorEmptyFillColor('#000000')
```

Setting the image editor's viewport size

You can change the size of the image editor's viewport using the `imageEditorViewportWidth()` and `imageEditorViewportHeight()` methods, which generate an aspect ratio to use across device sizes:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
->image()
->imageEditor()
->imageEditorViewportWidth('1920')
->imageEditorViewportHeight('1080')
```

Allowing users to crop images as a circle

You can allow users to crop images as a circle using the `circleCropper()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
->image()
->avatar()
->imageEditor()
->circleCropper()
```

This is perfectly accompanied by the `avatar()` method, which renders the images in a compact circle layout.

Cropping and resizing images without the editor

Filepond allows you to crop and resize images before they are uploaded, without the need for a separate editor. You can customize this behavior using the `imageCropAspectRatio()`, `imageResizeTargetHeight()` and `imageResizeTargetWidth()` methods. `imageResizeMode()` should be set for these methods to have an effect - either `force`, `cover`, or `contain`.

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
->image()
->imageResizeMode('cover')
->imageCropAspectRatio('16:9')
->imageResizeTargetWidth('1920')
->imageResizeTargetHeight('1080')
```

Altering the appearance of the file upload area

You may also alter the general appearance of the Filepond component. Available options for these methods are available on the [Filepond website](#).

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->imagePreviewHeight('250')
->loadingIndicatorPosition('left')
->panelAspectRatio('2:1')
->panelLayout('integrated')
->removeUploadedFileButtonPosition('right')
->uploadButtonPosition('left')
->uploadProgressIndicatorPosition('left')
```

Displaying files in a grid

You can use the [Filepond grid layout](#) by setting the `panelLayout()`:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->panelLayout('grid')
```

Reordering files

You can also allow users to re-order uploaded files using the `reorderable()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->reorderable()
```

When using this method, FilePond may add newly-uploaded files to the beginning of the list, instead of the end. To fix this, use the `appendFiles()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->reorderable()
->appendFiles()
```

Opening files in a new tab

You can add a button to open each file in a new tab with the `openable()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->openable()
```

Downloading files

If you wish to add a download button to each file instead, you can use the `downloadable()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->downloadable()
```

Previewing files

By default, some file types can be previewed in FilePond. If you wish to disable the preview for all files, you can use the `previewable(false)` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->previewable(false)
```

Moving files instead of copying when the form is submitted

By default, files are initially uploaded to Livewire's temporary storage directory, and then copied to the destination directory when the form is submitted. If you wish to move the files instead, providing that temporary uploads are stored on the same disk as permanent files, you can use the `moveFiles()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->moveFiles()
```

Preventing files from being stored permanently

If you wish to prevent files from being stored permanently when the form is submitted, you can use the `storeFiles(false)` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->storeFiles(false)
```

When the form is submitted, a temporary file upload object will be returned instead of a permanently stored file path. This is perfect for temporary files like imported CSVs.

Please be aware that images, video and audio files will not show the stored file name in the form's preview, unless you use `previewable(false)`. This is due to a limitation with the FilePond preview plugin.

Orienting images from their EXIF data

By default, FilePond will automatically orient images based on their EXIF data. If you wish to disable this behavior, you can use the `orientImagesFromExif(false)` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->orientImagesFromExif(false)
```

Hiding the remove file button

It is also possible to hide the remove uploaded file button by using `deletable(false)`:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->deletable(false)
```

Prevent file information fetching

While the form is loaded, it will automatically detect whether the files exist, what size they are, and what type of files they are. This is all done on the backend. When using remote storage with many files, this can be time-consuming. You can use the `fetchFileInfo(false)` method to disable this feature:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->fetchFileInfo(false)
```

Customizing the uploading message

You may customize the uploading message that is displayed in the form's submit button using the `uploadingMessage()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->uploadingMessage('Uploading attachment...')
```

File upload validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to file uploads.

Since Filament is powered by Livewire and uses its file upload system, you will want to refer to the default Livewire file upload validation rules in the `config/livewire.php` file as well. This also controls the 12MB file size maximum.

File type validation

You may restrict the types of files that may be uploaded using the `acceptedFileTypes()` method, and passing an array of MIME types.

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('document')
->acceptedFileTypes(['application/pdf'])
```

You may also use the `image()` method as shorthand to allow all image MIME types.

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('image')
->image()
```

File size validation

You may also restrict the size of uploaded files in kilobytes:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->minSize(512)
->maxSize(1024)
```

Number of files validation

You may customize the number of files that may be uploaded, using the `minFiles()` and `maxFiles()` methods:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
->multiple()
->minFiles(2)
->maxFiles(5)
```

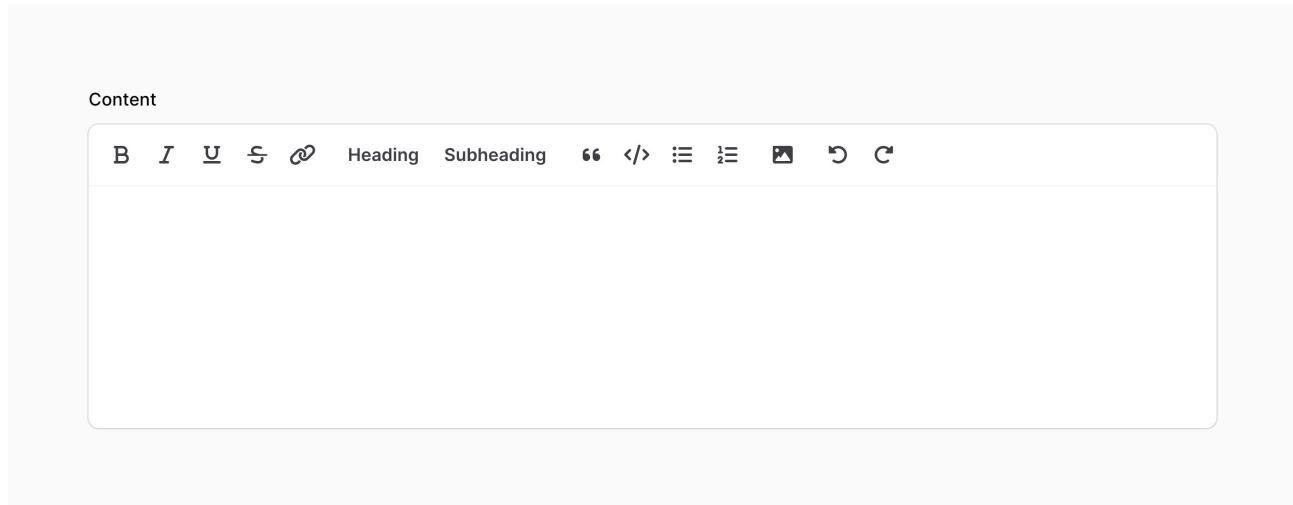
Rich Editor

Overview

The rich editor allows you to edit and preview HTML content, as well as upload images.

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
```



Customizing the toolbar buttons

You may set the toolbar buttons for the editor using the `toolbarButtons()` method. The options shown here are the defaults. In addition to these, `'h1'` is also available:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->toolbarButtons([
        'attachFiles',
        'blockquote',
        'bold',
        'bulletList',
        'codeBlock',
        'h2',
        'h3',
        'italic',
        'link',
        'orderedList',
        'redo',
        'strike',
        'underline',
        'undo',
    ])
)
```

Alternatively, you may disable specific buttons using the `disableToolbarButtons()` method:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->disableToolbarButtons([
        'blockquote',
        'strike',
    ])
```

To disable all toolbar buttons, set an empty array with `toolbarButtons([])` or use `disableAllToolbarButtons()`.

Uploading images to the editor

You may customize how images are uploaded using configuration methods:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->fileAttachmentsDisk('s3')
    ->fileAttachmentsDirectory('attachments')
    ->fileAttachmentsVisibility('private')
```

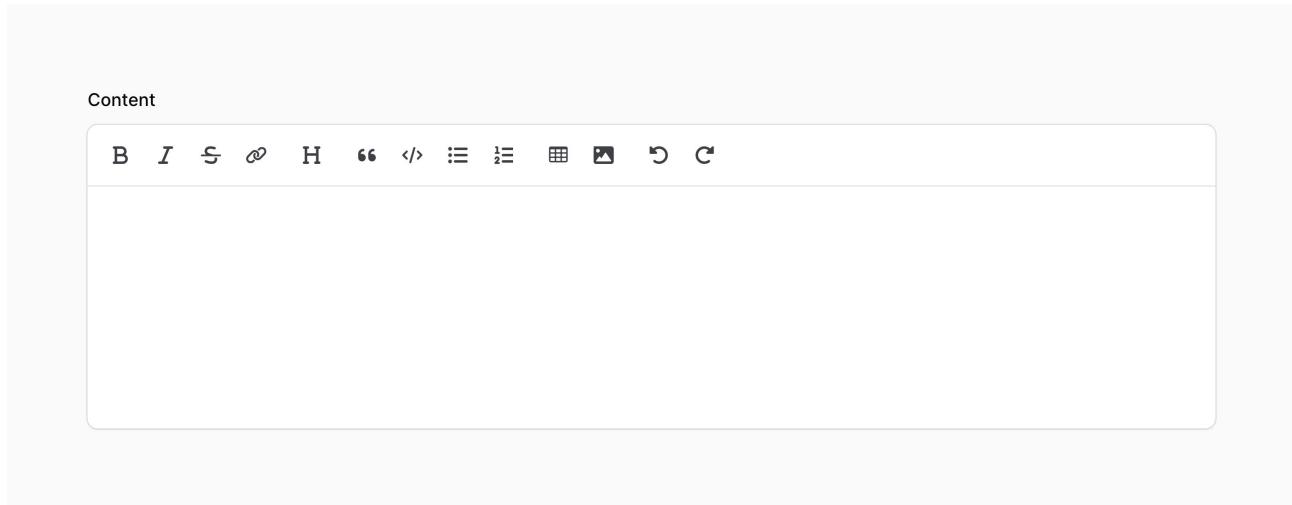
Markdown Editor

Overview

The markdown editor allows you to edit and preview markdown content, as well as upload images using drag and drop.

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
```



Customizing the toolbar buttons

You may set the toolbar buttons for the editor using the `toolbarButtons()` method. The options shown here are the defaults:

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
    ->toolbarButtons([
        'attachFiles',
        'blockquote',
        'bold',
        'bulletList',
        'codeBlock',
        'heading',
        'italic',
        'link',
        'orderedList',
        'redo',
        'strike',
        'table',
        'undo',
    ])
```

Alternatively, you may disable specific buttons using the `disableToolbarButtons()` method:

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
    ->disableToolbarButtons([
        'blockquote',
        'strike',
    ])
```

To disable all toolbar buttons, set an empty array with `toolbarButtons([])` or use `disableAllToolbarButtons()`.

Uploading images to the editor

You may customize how images are uploaded using configuration methods:

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
    ->fileAttachmentsDisk('s3')
    ->fileAttachmentsDirectory('attachments')
    ->fileAttachmentsVisibility('private')
```

Repeater

Overview

The repeater component allows you to output a JSON array of repeated form components.

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components>Select;
use Filament\Forms\Components\TextInput;

Repeater::make('members')
    ->schema([
        TextInput::make('name')->required(),
        Select::make('role')
            ->options([
                'member' => 'Member',
                'administrator' => 'Administrator',
                'owner' => 'Owner',
            ])
            ->required(),
    ])
    ->columns(2)
```

Members

↑↓	
Name*	Role*
Dan Harrin	Owner
↑↓	
Name*	Role*
Ryan Chandler	Administrator
↑↓	
Name*	Role*
Zep Fietje	Member
↑↓	
Name*	Role*
	Select an option
Add to members	

We recommend that you store repeater data with a `JSON` column in your database. Additionally, if you're using Eloquent, make sure that column has an `array` cast.

As evident in the above example, the component schema can be defined within the `schema()` method of the component:

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;

Repeater::make('members')
->schema([
    TextInput::make('name')->required(),
    // ...
])
```

If you wish to define a repeater with multiple schema blocks that can be repeated in any order, please use the [builder](#).

Setting empty default items

Repeaters may have a certain number of empty items created by default, using the `defaultItems()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->defaultItems(3)
```

Note that these default items are only created when the form is loaded without existing data. Inside [panel resources](#) this only works on Create Pages, as Edit Pages will always fill the data from the model.

Adding items

An action button is displayed below the repeater to allow the user to add a new item.

Setting the add action button's label

You may set a label to customize the text that should be displayed in the button for adding a repeater item, using the `addActionLabel()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->addActionLabel('Add member')
```

Preventing the user from adding items

You may prevent the user from adding items to the repeater using the `addable(false)` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->addable(false)
```

Deleting items

An action button is displayed on each item to allow the user to delete it.

Preventing the user from deleting items

You may prevent the user from deleting items from the repeater using the `deletable(false)` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->deletable(false)
```

Reordering items

A button is displayed on each item to allow the user to drag and drop to reorder it in the list.

Preventing the user from reordering items

You may prevent the user from reordering items from the repeater using the `reorderable(false)` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->reorderable(false)
```

Reordering items with buttons

You may use the `reorderableWithButtons()` method to enable reordering items with buttons to move the item up and down:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->reorderableWithButtons()
```

Members

Name*	Role*	Actions
Dan Harrin	Owner	
Ryan Chandler	Administrator	
Zep Fietje	Member	

Add to members

Preventing reordering with drag and drop

You may use the `reorderableWithDragAndDrop(false)` method to prevent items from being ordered with drag and drop:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->reorderableWithDragAndDrop(false)
```

Collapsing items

The repeater may be `collapsible()` to optionally hide content in long forms:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->schema([
        // ...
    ])
    ->collapsible()
```

You may also collapse all items by default:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
->schema([
    // ...
])
->collapsed()
```

Qualifications

Collapse all Expand all



Add to qualifications

Cloning items

You may allow repeater items to be duplicated using the `cloneable()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
->schema([
    // ...
])
->cloneable()
```

Qualifications

↑↓ ✖️ Name* <input type="text" value="Tailwind CSS Level 1"/>
↑↓ ✖️ Name* <input type="text" value="Alpine.js Level 1"/>
↑↓ ✖️ Name* <input type="text" value="Laravel Level 1"/>
↑↓ ✖️ Name* <input type="text" value="Livewire Level 1"/>
Add to qualifications

Integrating with an Eloquent relationship

If you're building a form inside your Livewire component, make sure you have set up the [form's model](#). Otherwise, Filament doesn't know which model to use to retrieve the relationship from.

You may employ the `relationship()` method of the `Repeater` to configure a `HasMany` relationship. Filament will load the item data from the relationship, and save it back to the relationship when the form is submitted. If a custom relationship name is not passed to `relationship()`, Filament will use the field name as the relationship name:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
```

When using `disabled()` with `relationship()`, ensure that `disabled()` is called before `relationship()`. This ensures that the `dehydrated()` call from within `relationship()` is not overridden by the call from `disabled()`:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->disabled()
    ->relationship()
    ->schema([
        // ...
    ])
```

Reordering items in a relationship

By default, `reordering` relationship repeater items is disabled. This is because your related model needs a `sort` column to store the order of related records. To enable reordering, you may use the `orderColumn()` method, passing in a name of the column on your related model to store the order in:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->orderColumn('sort')
```

If you use something like `spatie/eloquent-sortable` with an order column such as `order_column`, you may pass this in to `orderColumn()`:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->orderColumn('order_column')
```

Integrating with a `BelongsToMany` Eloquent relationship

There is a common misconception that using a `BelongsToMany` relationship with a repeater is as simple as using a `HasMany` relationship. This is not the case, as a `BelongsToMany` relationship requires a pivot table to store the relationship data. The repeater saves its data to the related model, not the pivot table. Therefore, if you want to map each repeater item to a row in the pivot table, you must use a `HasMany` relationship with a pivot model to use a repeater with a `BelongsToMany` relationship.

Imagine you have a form to create a new `Order` model. Each order belongs to many `Product` models, and each product belongs to many orders. You have a `order_product` pivot table to store the relationship data. Instead of using the `products` relationship with the repeater, you should create a new relationship called `orderProducts` on the `Order` model, and use that with the repeater:

```
use Illuminate\Database\Eloquent\Relations\HasMany;

public function orderProducts(): HasMany
{
    return $this->hasMany(OrderProduct::class);
}
```

If you don't already have an `OrderProduct` pivot model, you should create that, with inverse relationships to `Order` and `Product`:

```
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Relations\Pivot;

class OrderProduct extends Pivot
{
    public function order(): BelongsTo
    {
        return $this->belongsTo(Order::class);
    }

    public function product(): BelongsTo
    {
        return $this->belongsTo(Product::class);
    }
}
```

Please ensure that your pivot model has a primary key column, like `id`, to allow Filament to keep track of which repeater items have been created, updated and deleted.

Now you can use the `orderProducts` relationship with the repeater, and it will save the data to the `order_product` pivot table:

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components>Select;

Repeater::make('orderProducts')
    ->relationship()
    ->schema([
        Select::make('product_id')
            ->relationship('product', 'name')
            ->required(),
        // ...
    ])
```

Mutating related item data before filling the field

You may mutate the data for a related item before it is filled into the field using the

`mutateRelationshipDataBeforeFillUsing()` method. This method accepts a closure that receives the current item's data in a `$data` variable. You must return the modified array of data:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->mutateRelationshipDataBeforeFillUsing(function (array $data): array {
        $data['user_id'] = auth()->id();

        return $data;
    })
}
```

Mutating related item data before creating

You may mutate the data for a new related item before it is created in the database using the

`mutateRelationshipDataBeforeCreateUsing()` method. This method accepts a closure that receives the current item's data in a `$data` variable. You can choose to return either the modified array of data, or `null` to prevent the item from being created:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->mutateRelationshipDataBeforeCreateUsing(function (array $data): array {
        $data['user_id'] = auth()->id();

        return $data;
    })
}
```

Mutating related item data before saving

You may mutate the data for an existing related item before it is saved in the database using the

`mutateRelationshipDataBeforeSaveUsing()` method. This method accepts a closure that receives the current item's data in a `$data` variable. You can choose to return either the modified array of data, or `null` to prevent the item from being saved:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->mutateRelationshipDataBeforeSaveUsing(function (array $data): array {
        $data['user_id'] = auth()->id();

        return $data;
    })
}
```

Grid layout

You may organize repeater items into columns by using the `grid()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->schema([
        // ...
    ])
    ->grid(2)
```

The screenshot shows a 'Qualifications' repeater component. It consists of four items arranged in a 2x2 grid. Each item contains a 'Name*' field with the following values: 'Tailwind CSS Level 1', 'Alpine.js Level 1', 'Laravel Level 1', and 'Livewire Level 1'. Each item is accompanied by a red trash icon in the top right corner. A button labeled 'Add to qualifications' is located at the bottom center of the grid.

This method accepts the same options as the `columns()` method of the `grid`. This allows you to responsively customize the number of grid columns at various breakpoints.

Adding a label to repeater items based on their content

You may add a label for repeater items using the `itemLabel()` method. This method accepts a closure that receives the current item's data in a `$state` variable. You must return a string to be used as the item label:

```

use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components>Select;

Repeater::make('members')
    ->schema([
        TextInput::make('name')
            ->required()
            ->live(onBlur: true),
        Select::make('role')
            ->options([
                'member' => 'Member',
                'administrator' => 'Administrator',
                'owner' => 'Owner',
            ])
            ->required(),
    ])
    ->columns(2)
    ->itemLabel(fn (array $state): ?string => $state['name'] ?? null),

```

Any fields that you use from `$state` should be `live()` if you wish to see the item label update live as you use the form.

Name*	Role*
Dan Harrin	Owner
Ryan Chandler	Administrator
Zep Fietje	Member

Add to members

Simple repeaters with one field

You can use the `simple()` method to create a repeater with a single field, using a minimal design

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;

Repeater::make('invitations')
->simple(
    TextInput::make('email')
        ->email()
        ->required(),
)
)
```

The screenshot shows a 'Repeater' component titled 'Invitations'. It contains two items: 'dan@filamentphp.com' and 'ryan@filamentphp.com'. Each item has a small trash icon and a double-headed arrow icon to its right. Below the list is a button labeled 'Add to invitations'.

Instead of using a nested array to store data, simple repeaters use a flat array of values. This means that the data structure for the above example could look like this:

```
[  
    'invitations' => [  
        'dan@filamentphp.com',  
        'ryan@filamentphp.com',  
    ],  
],
```

Using `$get()` to access parent field values

All form components are able to use `$get()` and `$set()` to access another field's value. However, you might experience unexpected behavior when using this inside the repeater's schema.

This is because `$get()` and `$set()`, by default, are scoped to the current repeater item. This means that you are able to interact with another field inside that repeater item easily without knowing which repeater item the current form component belongs to.

The consequence of this is that you may be confused when you are unable to interact with a field outside the repeater. We use `.../` syntax to solve this problem - `$get('.../.../parent_field_name')`.

Consider your form has this data structure:

```
[  
    'client_id' => 1,  
  
    'repeater' => [  
        'item1' => [  
            'service_id' => 2,  
        ],  
    ],  
]
```

You are trying to retrieve the value of `client_id` from inside the repeater item.

`$get()` is relative to the current repeater item, so `$get('client_id')` is looking for `$get('repeater.item1.client_id')`.

You can use `.../` to go up a level in the data structure, so `$get('.../client_id')` is `$get('repeater.client_id')` and `$get('.../.../client_id')` is `$get('client_id')`.

The special case of `$get()` with no arguments, or `$get('')` or `$get('./')`, will always return the full data array for the current repeater item.

Repeater validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to repeaters.

Number of items validation

You can validate the minimum and maximum number of items that you can have in a repeater by setting the `minItems()` and `maxItems()` methods:

```
use Filament\Forms\Components\Repeater;  
  
Repeater::make('members')  
    ->schema([  
        // ...  
    ])  
    ->minItems(2)  
    ->maxItems(5)
```

Distinct state validation

In many cases, you will want to ensure some sort of uniqueness between repeater items. A couple of common examples could be:

- Ensuring that only one checkbox or toggle is activated at once across items in the repeater.
- Ensuring that an option may only be selected once across select, radio, checkbox list, or toggle buttons fields in a repeater.

You can use the `distinct()` method to validate that the state of a field is unique across all items in the repeater:

```
use Filament\Forms\Components\Checkbox;
use Filament\Forms\Components\Repeater;

Repeater::make('answers')
->schema([
    // ...
    Checkbox::make('is_correct')
        ->distinct(),
])
])
```

The behavior of the `distinct()` validation depends on the data type that the field handles

- If the field returns a boolean, like a `checkbox` or `toggle`, the validation will ensure that only one item has a value of `true`. There may be many fields in the repeater that have a value of `false`.
- Otherwise, for fields like a `select`, `radio`, `checkbox list`, or `toggle buttons`, the validation will ensure that each option may only be selected once across all items in the repeater.

Automatically fixing indistinct state

If you'd like to automatically fix indistinct state, you can use the `fixIndistinctState()` method:

```
use Filament\Forms\Components\Checkbox;
use Filament\Forms\Components\Repeater;

Repeater::make('answers')
->schema([
    // ...
    Checkbox::make('is_correct')
        ->fixIndistinctState(),
])
])
```

This method will automatically enable the `distinct()` and `live()` methods on the field.

Depending on the data type that the field handles, the behavior of the `fixIndistinctState()` adapts:

- If the field returns a boolean, like a `checkbox` or `toggle`, and one of the fields is enabled, Filament will automatically disable all other enabled fields on behalf of the user.
- Otherwise, for fields like a `select`, `radio`, `checkbox list`, or `toggle buttons`, when a user selects an option, Filament will automatically deselect all other usages of that option on behalf of the user.

Disabling options when they are already selected in another item

If you'd like to disable options in a `select`, `radio`, `checkbox list`, or `toggle buttons` when they are already selected in another item, you can use the `disableOptionsWhenSelectedInSiblingRepeaterItems()` method:

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components>Select;

Repeater::make('members')
->schema([
    Select::make('role')
        ->options([
            // ...
        ])
        ->disableOptionsWhenSelectedInSiblingRepeaterItems(),
])
])
```

This method will automatically enable the `distinct()` and `live()` methods on the field.

Customizing the repeater item actions

This field uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to [customize it](#). The following methods are available to customize the actions:

- `addAction()`
- `cloneAction()`
- `collapseAction()`
- `collapseAllAction()`
- `deleteAction()`
- `expandAction()`
- `expandAllAction()`
- `moveDownAction()`
- `moveUpAction()`
- `reorderAction()`

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->collapseAllAction(
        fn (Action $action) => $action->label('Collapse all members'),
    )
```

Confirming repeater actions with a modal

You can confirm actions with a modal by using the `requiresConfirmation()` method on the action object. You may use any [modal customization method](#) to change its content and behavior:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->schema([
        // ...
    ])
    ->deleteAction(
        fn (Action $action) => $action->requiresConfirmation(),
    )
```

The `collapseAction()`, `collapseAllAction()`, `expandAction()`, `expandAllAction()` and `reorderAction()` methods do not support confirmation modals, as clicking their buttons does not make the network request that is required to show the modal.

Adding extra item actions to a repeater

You may add new [action buttons](#) to the header of each repeater item by passing `Action` objects into `extraItemActions()`:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Mail;

Repeater::make('members')
->schema([
    TextInput::make('email')
        ->label('Email address')
        ->email(),
    // ...
])
->extraItemActions([
    Action::make('sendEmail')
        ->icon('heroicon-m-envelope')
        ->action(function (array $arguments, Repeater $component): void {
            $itemData = $component->getItemState($arguments['item']);

            Mail::to($itemData['email'])
                ->send(
                    // ...
                );
        }),
])
])
```

In this example, `$arguments['item']` gives you the ID of the current repeater item. You can validate the data in that repeater item using the `getItemState()` method on the repeater component. This method returns the validated data for the item. If the item is not valid, it will cancel the action and show an error message for that item in the form.

If you want to get the raw data from the current item without validating it, you can use `$component->getRawItemState($arguments['item'])` instead.

If you want to manipulate the raw data for the entire repeater, for example, to add, remove or modify items, you can use `$component->getState()` to get the data, and `$component->state($state)` to set it again:

```
use Illuminate\Support\Str;

// Get the raw data for the entire repeater
$state = $component->getState();

// Add an item, with a random UUID as the key
$state[Str::uuid()] = [
    'email' => auth()->user()->email,
];

// Set the new data for the repeater
$component->state($state);
```

Testing repeaters

Internally, repeaters generate UUIDs for items to keep track of them in the Livewire HTML easier. This means that when you are testing a form with a repeater, you need to ensure that the UUIDs are consistent between the form and the test.

This can be tricky, and if you don't do it correctly, your tests can fail as the tests are expecting a UUID, not a numeric key.

However, since Livewire doesn't need to keep track of the UUIDs in a test, you can disable the UUID generation and replace them with numeric keys, using the `Repeater::fake()` method at the start of your test:

```
use Filament\Forms\Components\Repeater;
use function Pest\Livewire\livewire;

$undoRepeaterFake = Repeater::fake();

livewire(EditPost::class, ['record' => $post])
->assertFormSet([
    'quotes' => [
        [
            'content' => 'First quote',
        ],
        [
            'content' => 'Second quote',
        ],
        // ...
    ],
]);
}

$undoRepeaterFake();
```

You may also find it useful to access test the number of items in a repeater by passing a function to the `assertFormSet()` method:

```
use Filament\Forms\Components\Repeater;
use function Pest\Livewire\livewire;

$undoRepeaterFake = Repeater::fake();

livewire(EditPost::class, ['record' => $post])
->assertFormSet(function (array $state) {
    expect($state['quotes'])
        ->toHaveCount(2);
});

$undoRepeaterFake();
```

Builder

Overview

Similar to a [repeater](#), the builder component allows you to output a JSON array of repeated form components. Unlike the repeater, which only defines one form schema to repeat, the builder allows you to define different schema "blocks", which you can repeat in any order. This makes it useful for building more advanced array structures.

The primary use of the builder component is to build web page content using predefined blocks. This could be content for a marketing website, or maybe even fields in an online form. The example below defines multiple blocks for different elements in the page content. On the frontend of your website, you could loop through each block in the JSON and format it how you wish.

```
use Filament\Forms\Components\Builder;
use Filament\Forms\Components\FileUpload;
use Filament\Forms\Components>Select;
use Filament\Forms\Components\Textarea;
use Filament\Forms\Components\TextInput;

Builder::make('content')
    ->blocks([
        Builder\Block::make('heading')
            ->schema([
                TextInput::make('content')
                    ->label('Heading')
                    ->required(),
                Select::make('level')
                    ->options([
                        'h1' => 'Heading 1',
                        'h2' => 'Heading 2',
                        'h3' => 'Heading 3',
                        'h4' => 'Heading 4',
                        'h5' => 'Heading 5',
                        'h6' => 'Heading 6',
                    ])
                    ->required(),
            ])
            ->columns(2),
        Builder\Block::make('paragraph')
            ->schema([
                Textarea::make('content')
                    ->label('Paragraph')
                    ->required(),
            ]),
        Builder\Block::make('image')
            ->schema([
                FileUpload::make('url')
                    ->label('Image')
                    ->image()
                    ->required(),
                TextInput::make('alt')
                    ->label('Alt text')
                    ->required(),
            ]),
    ])
])
```

Content

Heading 1 Delete

Heading* **Level***

Heading 2

Paragraph 2 Delete

Paragraph*

 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl quis nunc. Donec euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl

Image 3 Delete

Image*

 Drag & Drop your files or [Browse](#)

Alt text*

 Lorem ipsum dolor sit amet

Paragraph 4 Delete

Paragraph*

 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl quis nunc. Donec euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl

Image 5 Delete

Image*

 Drag & Drop your files or [Browse](#)

Alt text*

 Lorem ipsum dolor sit amet

[Add to content](#)

We recommend that you store builder data with a `JSON` column in your database. Additionally, if you're using Eloquent, make sure that column has an `array` cast.

As evident in the above example, blocks can be defined within the `blocks()` method of the component. Blocks are `[Builder\Block]` objects, and require a unique name, and a component schema:

```
use Filament\Forms\Components\Builder;
use Filament\Forms\Components\TextInput;

Builder::make('content')
->blocks([
    Builder\Block::make('heading')
        ->schema([
            TextInput::make('content')->required(),
            // ...
        ]),
        // ...
    ])
])
```

Setting a block's label

By default, the label of the block will be automatically determined based on its name. To override the block's label, you may use the `label()` method. Customizing the label in this way is useful if you wish to use a [translation string for localization](#):

```
use Filament\Forms\Components\Builder;

Builder\Block::make('heading')
->label(__('blocks.heading'))
```

Labelling builder items based on their content

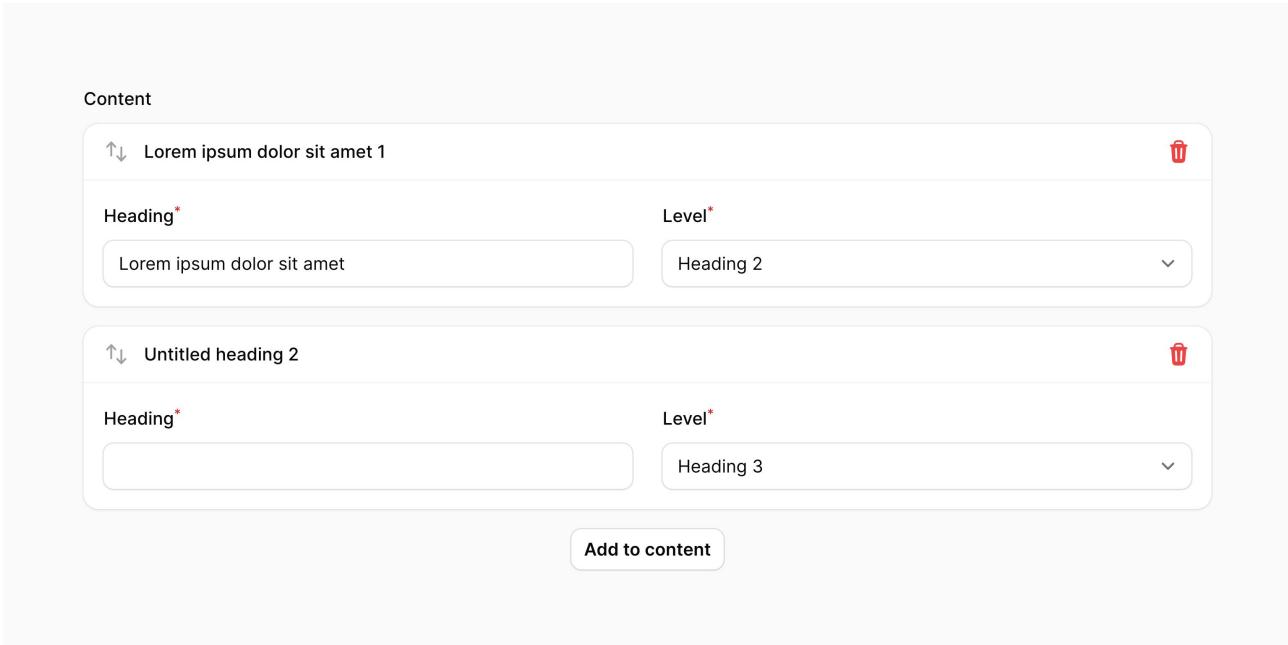
You may add a label for a builder item using the same `label()` method. This method accepts a closure that receives the item's data in a `$state` variable. If `$state` is null, you should return the block label that should be displayed in the block picker. Otherwise, you should return a string to be used as the item label:

```
use Filament\Forms\Components\Builder;
use Filament\Forms\Components\TextInput;

Builder\Block::make('heading')
->schema([
    TextInput::make('content')
        ->live(onBlur: true)
        ->required(),
    // ...
])
->label(function (?array $state): string {
    if ($state === null) {
        return 'Heading';
    }

    return $state['content'] ?? 'Untitled heading';
})
```

Any fields that you use from `$state` should be `live()` if you wish to see the item label update live as you use the form.



Numbering builder items

By default, items in the builder have a number next to their label. You may disable this using the `blockNumbers(false)` method:

```
use Filament\Forms\Components\Builder;

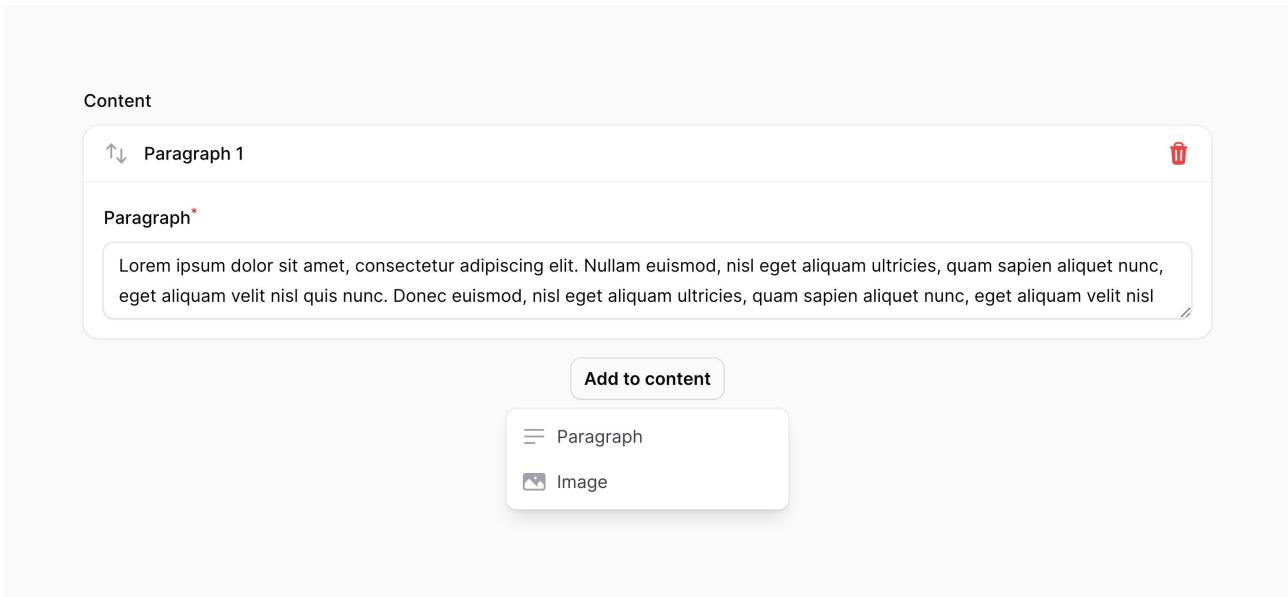
Builder::make('content')
    ->blocks([
        // ...
    ])
    ->blockNumbers(false)
```

Setting a block's icon

Blocks may also have an `icon`, which is displayed next to the label. You can add an icon by passing its name to the `icon()` method:

```
use Filament\Forms\Components\Builder;

Builder\Block::make('paragraph')
    ->icon('heroicon-m-bars-3-bottom-left')
```



Adding icons to the header of blocks

By default, blocks in the builder don't have an icon next to the header label, just in the dropdown to add new blocks. You may enable this using the `blockIcons()` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->blockIcons()
```

Adding items

An action button is displayed below the builder to allow the user to add a new item.

Setting the add action button's label

You may set a label to customize the text that should be displayed in the button for adding a builder item, using the `addActionLabel()` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->addActionLabel('Add a new block')
```

Preventing the user from adding items

You may prevent the user from adding items to the builder using the `addable(false)` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->addable(false)
```

Deleting items

An action button is displayed on each item to allow the user to delete it.

Preventing the user from deleting items

You may prevent the user from deleting items from the builder using the `deletable(false)` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->deletable(false)
```

Reordering items

A button is displayed on each item to allow the user to drag and drop to reorder it in the list.

Preventing the user from reordering items

You may prevent the user from reordering items from the builder using the `reorderable(false)` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->reorderable(false)
```

Reordering items with buttons

You may use the `reorderableWithButtons()` method to enable reordering items with buttons to move the item up and down:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->reorderableWithButtons()
```

Content

The screenshot shows a content editor interface with three sections, each containing a paragraph of placeholder text. Each section has a set of small arrows at the top right for reordering, a trash icon at the top right, and a larger trash icon at the bottom right. The sections are labeled 'Paragraph 1', 'Paragraph 2', and 'Paragraph 3'. A central button labeled 'Add to content' is located below the third section.

Preventing reordering with drag and drop

You may use the `reorderableWithDragAndDrop(false)` method to prevent items from being ordered with drag and drop:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->reorderableWithDragAndDrop(false)
```

Collapsing items

The builder may be `collapsible()` to optionally hide content in long forms:

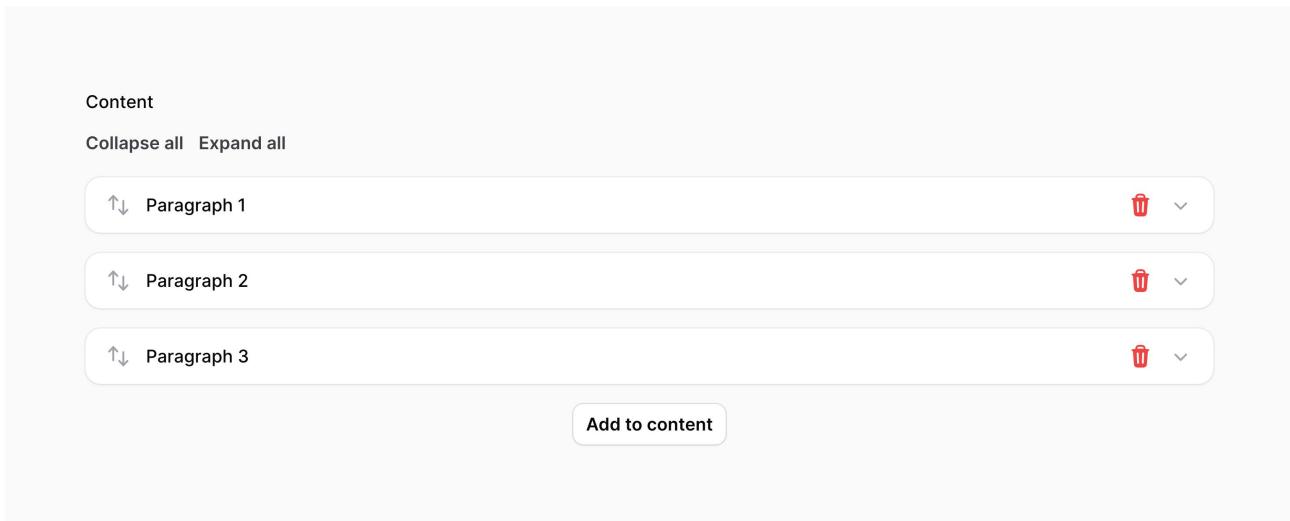
```
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->collapsible()
```

You may also collapse all items by default:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->collapsed()
```



Cloning items

You may allow builder items to be duplicated using the `cloneable()` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->cloneable()
```

The screenshot shows the Filament 3.x Content editor interface. It displays three paragraphs of text, each with a title, a rich text editor, and a set of block-level controls (copy, move, delete). The paragraphs are labeled "Paragraph 1", "Paragraph 2", and "Paragraph 3". Each paragraph contains placeholder text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl quis nunc. Donec euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc, eget aliquam velit nisl". A central "Add to content" button is visible at the bottom.

Customizing the block picker

Changing the number of columns in the block picker

The block picker has only 1 column. You may customize it by passing a number of columns to `blockPickerColumns()`:

```
use Filament\Forms\Components\Builder;

Builder::make()
->blockPickerColumns(2)
->blocks([
    // ...
])
```

This method can be used in a couple of different ways:

- You can pass an integer like `blockPickerColumns(2)`. This integer is the number of columns used on the `lg` breakpoint and higher. All smaller devices will have just 1 column.
- You can pass an array, where the key is the breakpoint and the value is the number of columns. For example, `blockPickerColumns(['md' => 2, 'xl' => 4])` will create a 2 column layout on medium devices, and a 4 column layout on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

Breakpoints (`sm`, `md`, `lg`, `xl`, `2xl`) are defined by Tailwind, and can be found in the [Tailwind documentation](#).

Increasing the width of the block picker

When you increase the number of columns, the width of the dropdown should increase incrementally to handle the additional columns. If you'd like more control, you can manually set a maximum width for the dropdown using the `blockPickerWidth()` method. Options correspond to [Tailwind's max-width scale](#). The options are `xs`, `sm`, `md`, `lg`, `xl`, `2xl`, `3xl`, `4xl`, `5xl`, `6xl`, `7xl`:

```
use Filament\Forms\Components\Builder;

Builder::make()
    ->blockPickerColumns(3)
    ->blockPickerWidth('2xl')
    ->blocks([
        // ...
    ])
```

Limiting the number of times a block can be used

By default, each block can be used in the builder an unlimited number of times. You may limit this using the `maxItems()` method on a block:

```
use Filament\Forms\Components\Builder;

Builder\Block::make('heading')
    ->schema([
        // ...
    ])
    ->maxItems(1)
```

Builder validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to builders.

Number of items validation

You can validate the minimum and maximum number of items that you can have in a builder by setting the `minItems()` and `maxItems()` methods:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->minItems(1)
    ->maxItems(5)
```

Using `$get()` to access parent field values

All form components are able to use `$get()` and `$set()` to access another field's value. However, you might experience unexpected behavior when using this inside the builder's schema.

This is because `$get()` and `$set()`, by default, are scoped to the current builder item. This means that you are able to interact with another field inside that builder item easily without knowing which builder item the current form component belongs to.

The consequence of this is that you may be confused when you are unable to interact with a field outside the builder. We use `.../` syntax to solve this problem - `$get('.../parent_field_name')`.

Consider your form has this data structure:

```
[  
    'client_id' => 1,  
  
    'builder' => [  
        'item1' => [  
            'service_id' => 2,  
        ],  
    ],  
]
```

You are trying to retrieve the value of `client_id` from inside the builder item.

`$get()` is relative to the current builder item, so `$get('client_id')` is looking for `$get('builder.item1.client_id')`.

You can use `.../` to go up a level in the data structure, so `$get('.../client_id')` is `$get('builder.client_id')` and `$get('.../..../client_id')` is `$get('client_id')`.

The special case of `$get()` with no arguments, or `$get('')` or `$get('..')`, will always return the full data array for the current builder item.

Customizing the builder item actions

This field uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to [customize it](#). The following methods are available to customize the actions:

- `addAction()`
- `addBetweenAction()`
- `cloneAction()`
- `collapseAction()`
- `collapseAllAction()`
- `deleteAction()`
- `expandAction()`
- `expandAllAction()`
- `moveDownAction()`
- `moveUpAction()`
- `reorderAction()`

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->collapseAllAction(
    fn (Action $action) => $action->label('Collapse all content'),
)
```

Confirming builder actions with a modal

You can confirm actions with a modal by using the `requiresConfirmation()` method on the action object. You may use any [modal customization method](#) to change its content and behavior:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Builder;

Builder::make('content')
->blocks([
    // ...
])
->deleteAction(
    fn (Action $action) => $action->requiresConfirmation(),
)
```

The `addAction()`, `addBetweenAction()`, `collapseAction()`, `collapseAllAction()`, `expandAction()`, `expandAllAction()` and `reorderAction()` methods do not support confirmation modals, as clicking their buttons does not make the network request that is required to show the modal.

Adding extra item actions to a builder

You may add new [action buttons](#) to the header of each builder item by passing `Action` objects into `extraItemActions()`:

```

use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Builder;
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Mail;

Builder::make('content')
    ->blocks([
        Builder\Block::make('contactDetails')
            ->schema([
                TextInput::make('email')
                    ->label('Email address')
                    ->email()
                    ->required(),
                    // ...
                ],
                // ...
            ])
        ->extraItemActions([
            Action::make('sendEmail')
                ->icon('heroicon-m-square-2-stack')
                ->action(function (array $arguments, Builder $component): void {
                    $itemData = $component->getItemState($arguments['item']);

                    Mail::to($itemData['email'])
                        ->send(
                            // ...
                        );
                }),
            ],
        ])
    )
]
)

```

In this example, `$arguments['item']` gives you the ID of the current builder item. You can validate the data in that builder item using the `getItemState()` method on the builder component. This method returns the validated data for the item. If the item is not valid, it will cancel the action and show an error message for that item in the form.

If you want to get the raw data from the current item without validating it, you can use `$component->getRawItemState($arguments['item'])` instead.

If you want to manipulate the raw data for the entire builder, for example, to add, remove or modify items, you can use `$component->getState()` to get the data, and `$component->state($state)` to set it again:

```

use Illuminate\Support\Str;

// Get the raw data for the entire builder
$state = $component->getState();

// Add an item, with a random UUID as the key
$state[Str::uuid()] = [
    'type' => 'contactDetails',
    'data' => [
        'email' => auth()->user()->email,
    ],
];

// Set the new data for the builder
$component->state($state);

```

Previewing blocks

If you prefer to render read-only previews in the builder instead of the blocks' forms, you can use the `blockPreviews()` method. This will render each block's `preview()` instead of the form. Block data will be passed to the preview Blade view in a variable with the same name:

```

use Filament\Forms\Components\Builder;
use Filament\Forms\Components\Builder\Block;
use Filament\Forms\Components\TextInput;

Builder::make('content')
    ->blockPreviews()
    ->blocks([
        Block::make('heading')
            ->schema([
                TextInput::make('text')
                    ->placeholder('Default heading'),
            ])
            ->preview('filament.content.block-previews.heading'),
    ])
]

```

In `/resources/views/filament/content/block-previews/heading.blade.php`, you can access the block data like so:

```

<h1>
    {{ $text ?? 'Default heading' }}
</h1>

```

Interactive block previews

By default, preview content is not interactive, and clicking it will open the Edit modal for that block to manage its settings. If you have links and buttons that you'd like to remain interactive in the block previews, you can use the `areInteractive: true` argument of the `blockPreviews()` method:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blockPreviews(areInteractive: true)
    ->blocks([
        //
    ])
)
```

Testing builders

Internally, builders generate UUIDs for items to keep track of them in the Livewire HTML easier. This means that when you are testing a form with a builder, you need to ensure that the UUIDs are consistent between the form and the test. This can be tricky, and if you don't do it correctly, your tests can fail as the tests are expecting a UUID, not a numeric key.

However, since Livewire doesn't need to keep track of the UUIDs in a test, you can disable the UUID generation and replace them with numeric keys, using the `Builder::fake()` method at the start of your test:

```
use Filament\Forms\Components\Builder;
use function Pest\Livewire\livewire;

$undoBuilderFake = Builder::fake();

livewire(EditPost::class, ['record' => $post])
    ->assertFormSet([
        'content' => [
            [
                'type' => 'heading',
                'data' => [
                    'content' => 'Hello, world!',
                    'level' => 'h1',
                ],
            ],
            [
                'type' => 'paragraph',
                'data' => [
                    'content' => 'This is a test post.',
                ],
            ],
            [
                // ...
            ],
        ],
    ]);
}

$undoBuilderFake();
```

You may also find it useful to access test the number of items in a repeater by passing a function to the `assertFormSet()` method:

```
use Filament\Forms\Components\Builder;
use function Pest\Livewire\livewire;

$undoBuilderFake = Builder::fake();

livewire(EditPost::class, ['record' => $post])
->assertFormSet(function (array $state) {
    expect($state['content'])
        ->toHaveCount(2);
});

$undoBuilderFake();
```

Tags Input

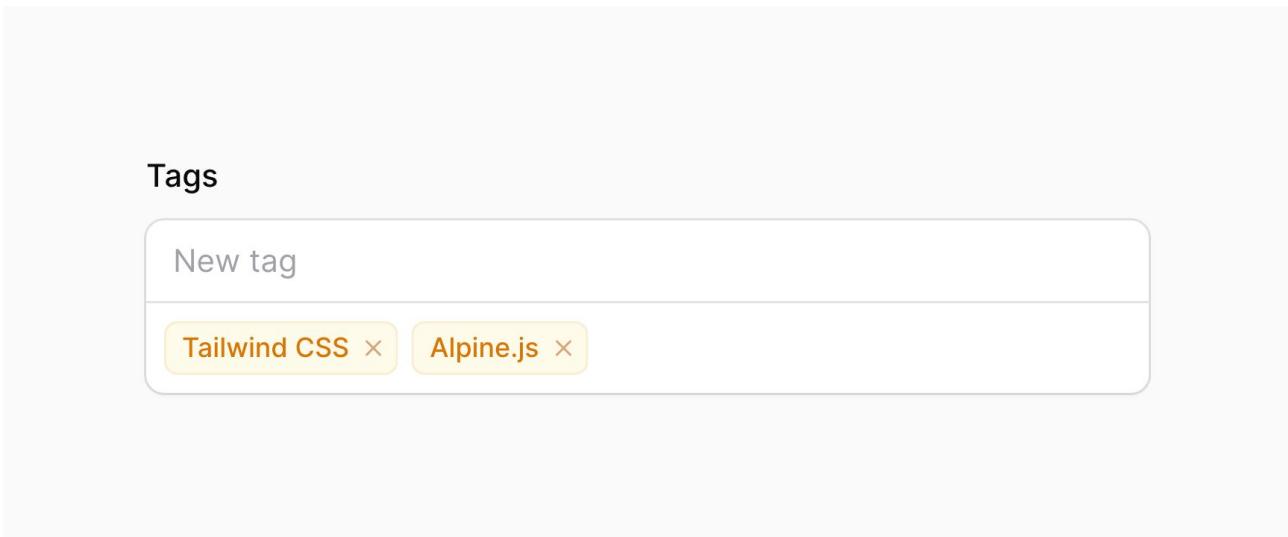
Overview

The tags input component allows you to interact with a list of tags.

By default, tags are stored in JSON:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
```



If you're saving the JSON tags using Eloquent, you should be sure to add an `array` `cast` to the model property:

```
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $casts = [
        'tags' => 'array',
    ];

    // ...
}
```

Filament also supports `spatie/laravel-tags`. See our [plugin documentation](#) for more information.

Comma-separated tags

You may allow the tags to be stored in a separated string, instead of JSON. To set this up, pass the separating character to the `separator()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->separator(',',')
```

Autocompleting tag suggestions

Tags inputs may have autocomplete suggestions. To enable this, pass an array of suggestions to the `suggestions()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->suggestions([
        'tailwindcss',
        'alpinejs',
        'laravel',
        'livewire',
    ])
])
```

Defining split keys

Split keys allow you to map specific buttons on your user's keyboard to create a new tag. By default, when the user presses "Enter", a new tag is created in the input. You may also define other keys to create new tags, such as "Tab" or " ". To do this, pass an array of keys to the `splitKeys()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->splitKeys(['Tab', ' '])
```

You can [read more about possible options for keys](#).

Adding a prefix and suffix to individual tags

You can add prefix and suffix to tags without modifying the real state of the field. This can be useful if you need to show presentational formatting to users without saving it. This is done with the `tagPrefix()` or `tagSuffix()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('percentages')
    ->tagSuffix('%')
```

Reordering tags

You can allow the user to reorder tags within the field using the `reorderable()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->reorderable()
```

Changing the color of tags

You can change the color of the tags by passing a color to the `color()` method. It may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->color('danger')
```

Tags validation

You may add validation rules for each tag by passing an array of rules to the `nestedRecursiveRules()` method:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
    ->nestedRecursiveRules([
        'min:3',
        'max:255',
    ])
```

Textarea

Overview

The textarea allows you to interact with a multi-line string:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
```

Description

`Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, quam sapien aliquet nunc,`

Resizing the textarea

You may change the size of the textarea by defining the `rows()` and `cols()` methods:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
    ->rows(10)
    ->cols(20)
```

Autosizing the textarea

You may allow the textarea to automatically resize to fit its content by setting the `autosize()` method:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
    ->autosize()
```

Making the field read-only

Not to be confused with [disabling the field](#), you may make the field "read-only" using the `readOnly()` method:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
->readOnly()
```

There are a few differences, compared to `disabled()`:

- When using `readOnly()`, the field will still be sent to the server when the form is submitted. It can be mutated with the browser console, or via JavaScript. You can use `dehydrated(false)` to prevent this.
- There are no styling changes, such as less opacity, when using `readOnly()`.
- The field is still focusable when using `readOnly()`.

Textarea validation

As well as all rules listed on the [validation](#) page, there are additional rules that are specific to textareas.

Length validation

You may limit the length of the textarea by setting the `minLength()` and `maxLength()` methods. These methods add both frontend and backend validation:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
->minLength(2)
->maxLength(1024)
```

You can also specify the exact length of the textarea by setting the `length()`. This method adds both frontend and backend validation:

```
use Filament\Forms\Components\Textarea;

Textarea::make('question')
->length(100)
```

Key Value

Overview

The key-value field allows you to interact with one-dimensional JSON object:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
```

Meta

Key	Value	
description	Filament is a collection of Laravel packages	
og:type	website	
og:site_name	Filament	
Add row		

If you're saving the data in Eloquent, you should be sure to add an `array` `cast` to the model property:

```
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $casts = [
        'meta' => 'array',
    ];

    // ...
}
```

Adding rows

An action button is displayed below the field to allow the user to add a new row.

Setting the add action button's label

You may set a label to customize the text that should be displayed in the button for adding a row, using the `addActionLabel()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->addActionLabel('Add property')
```

Preventing the user from adding rows

You may prevent the user from adding rows using the `addable(false)` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->addable(false)
```

Deleting rows

An action button is displayed on each item to allow the user to delete it.

Preventing the user from deleting rows

You may prevent the user from deleting rows using the `deletable(false)` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->deletable(false)
```

Editing keys

Customizing the key fields' label

You may customize the label for the key fields using the `keyLabel()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->keyLabel('Property name')
```

Adding key field placeholders

You may also add placeholders for the key fields using the `keyPlaceholder()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->keyPlaceholder('Property name')
```

Preventing the user from editing keys

You may prevent the user from editing keys using the `editableKeys(false)` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->editableKeys(false)
```

Editing values

Customizing the value fields' label

You may customize the label for the value fields using the `valueLabel()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->valueLabel('Property value')
```

Adding value field placeholders

You may also add placeholders for the value fields using the `valuePlaceholder()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->valuePlaceholder('Property value')
```

Preventing the user from editing values

You may prevent the user from editing values using the `editableValues(false)` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->editableValues(false)
```

Reordering rows

You can allow the user to reorder rows within the table using the `reorderable()` method:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->reorderable()
```

Meta		
Key	Value	
↑↓ description	Filament is a collection of Laravel packages	
↑↓ og:type	website	
↑↓ og:site_name	Filament	
Add row		

Customizing the key-value action objects

This field uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to [customize it](#). The following methods are available to customize the actions:

- `addAction()`
- `deleteAction()`
- `reorderAction()`

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->deleteAction(
    fn (Action $action) => $action->icon('heroicon-m-x-mark'),
)
```

Color Picker

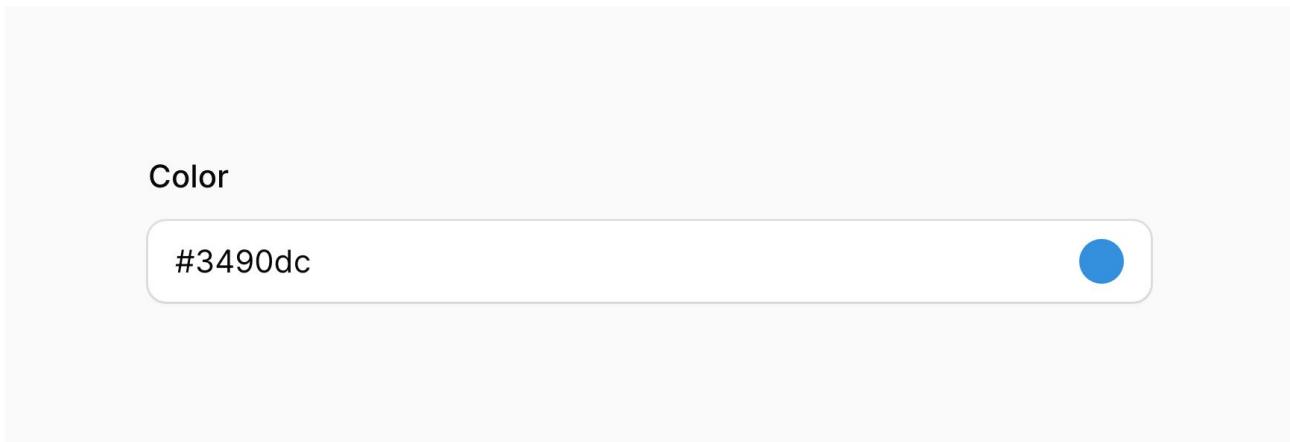
Overview

The color picker component allows you to pick a color in a range of formats.

By default, the component uses HEX format:

```
use Filament\Forms\Components\ColorPicker;

ColorPicker::make('color')
```



Setting the color format

While HEX format is used by default, you can choose which color format to use:

```
use Filament\Forms\Components\ColorPicker;

ColorPicker::make('hsl_color')
->hsl()

ColorPicker::make('rgb_color')
->rgb()

ColorPicker::make('rgba_color')
->rgba()
```

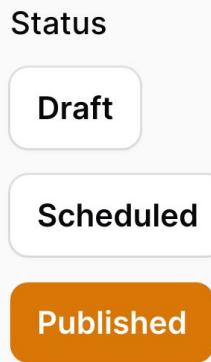
Toggle Buttons

Overview

The toggle buttons input provides a group of buttons for selecting a single value, or multiple values, from a list of predefined options:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('status')
    ->options([
        'draft' => 'Draft',
        'scheduled' => 'Scheduled',
        'published' => 'Published'
    ])
)
```



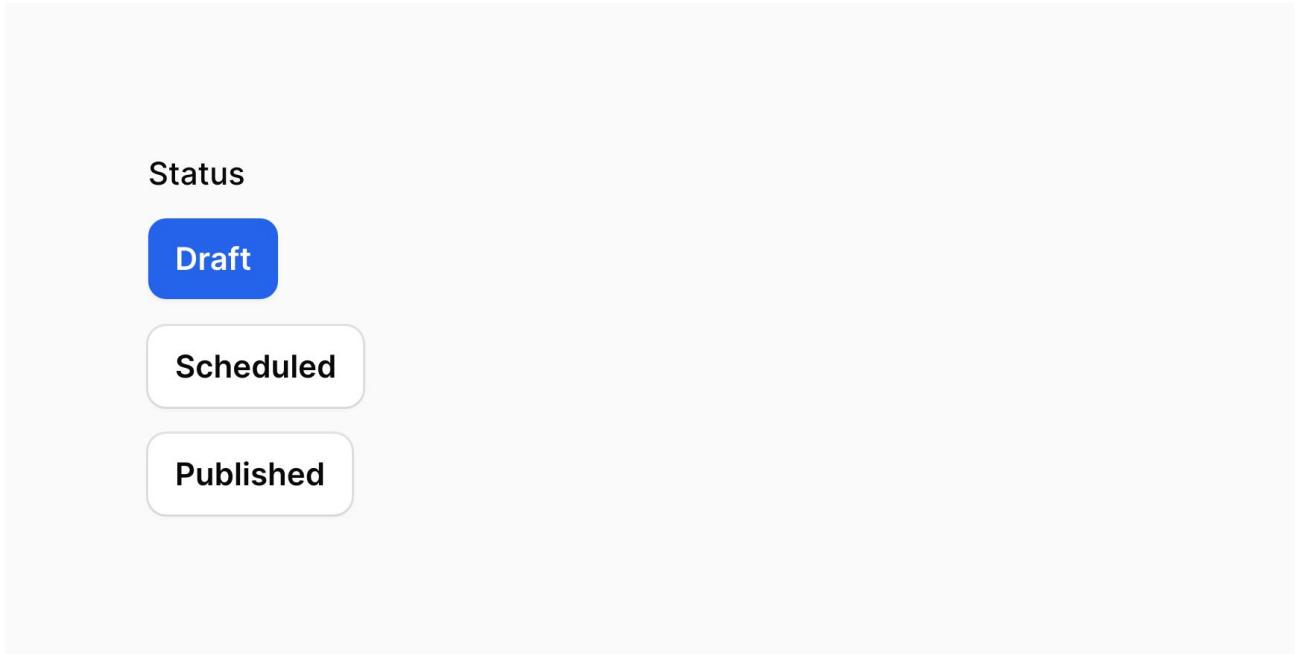
Changing the color of option buttons

You can change the color of the option buttons using the `colors()` method. Each key in the array should correspond to an option value, and the value may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('status')
    ->options([
        'draft' => 'Draft',
        'scheduled' => 'Scheduled',
        'published' => 'Published'
    ])
    ->colors([
        'draft' => 'info',
        'scheduled' => 'warning',
        'published' => 'success',
    ])
)
```

If you are using an enum for the options, you can use the [HasColor interface](#) to define colors instead.



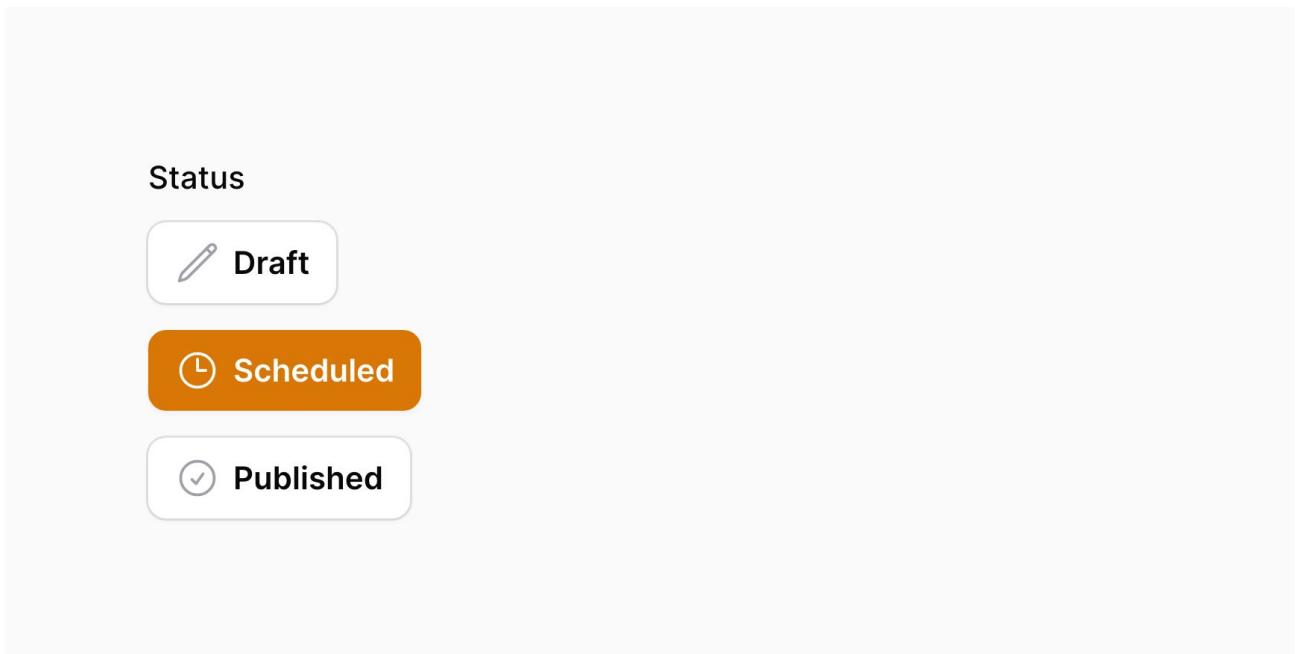
Adding icons to option buttons

You can add [icon](#) to the option buttons using the `[icons()]` method. Each key in the array should correspond to an option value, and the value may be any valid [Blade icon](#):

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('status')
    ->options([
        'draft' => 'Draft',
        'scheduled' => 'Scheduled',
        'published' => 'Published'
    ])
    ->icons([
        'draft' => 'heroicon-o-pencil',
        'scheduled' => 'heroicon-o-clock',
        'published' => 'heroicon-o-check-circle',
    ])
)
```

If you are using an enum for the options, you can use the [HasIcon](#) interface to define icons instead.



Boolean options

If you want a simple boolean toggle button group, with "Yes" and "No" options, you can use the [boolean\(\)](#) method:

```
ToggleButtons::make('feedback')
    ->label('Like this post?')
    ->boolean()
```

The options will have [colors](#) and [icons](#) set up automatically, but you can override these with [colors\(\)](#) or [icons\(\)](#).

Like this post?

Yes

No

Positioning the options inline with each other

You may wish to display the options `inline()` with each other:

```
ToggleButtons::make('feedback')
    ->label('Like this post?')
    ->boolean()
    ->inline()
```

Like this post?

Yes

No

Grouping option buttons

You may wish to group option buttons together so they are more compact, using the `grouped()` method. This also makes them appear horizontally inline with each other:

```
ToggleButtons::make('feedback')
    ->label('Like this post?')
    ->boolean()
    ->grouped()
```

Like this post?

✓ Yes

✗ No

Selecting multiple buttons

The `multiple()` method on the `ToggleButtons` component allows you to select multiple values from the list of options:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('technologies')
    ->multiple()
    ->options([
        'tailwind' => 'Tailwind CSS',
        'alpine' => 'Alpine.js',
        'laravel' => 'Laravel',
        'livewire' => 'Laravel Livewire',
    ])
)
```

Technologies

Tailwind CSS

Alpine.js

Laravel

Laravel Livewire

These options are returned in JSON format. If you're saving them using Eloquent, you should be sure to add an `array` [cast](#) to the model property:

```
use Illuminate\Database\Eloquent\Model;

class App extends Model
{
    protected $casts = [
        'technologies' => 'array',
    ];

    // ...
}
```

Splitting options into columns

You may split options into columns by using the `columns()` method:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('technologies')
    ->options([
        // ...
    ])
    ->columns(2)
```

Technologies

Tailwind CSS

Laravel

Alpine.js

Laravel Livewire

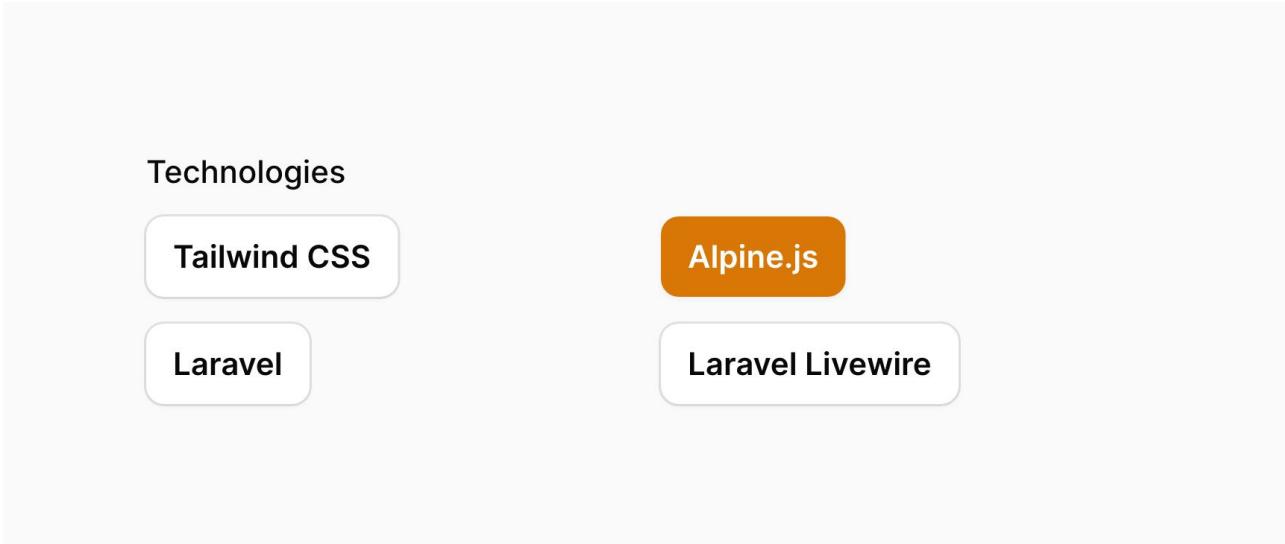
This method accepts the same options as the `columns()` method of the [grid](#). This allows you to responsively customize the number of columns at various breakpoints.

Setting the grid direction

By default, when you arrange buttons into columns, they will be listed in order vertically. If you'd like to list them horizontally, you may use the `gridDirection('row')` method:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('technologies')
    ->options([
        // ...
    ])
    ->columns(2)
    ->gridDirection('row')
```

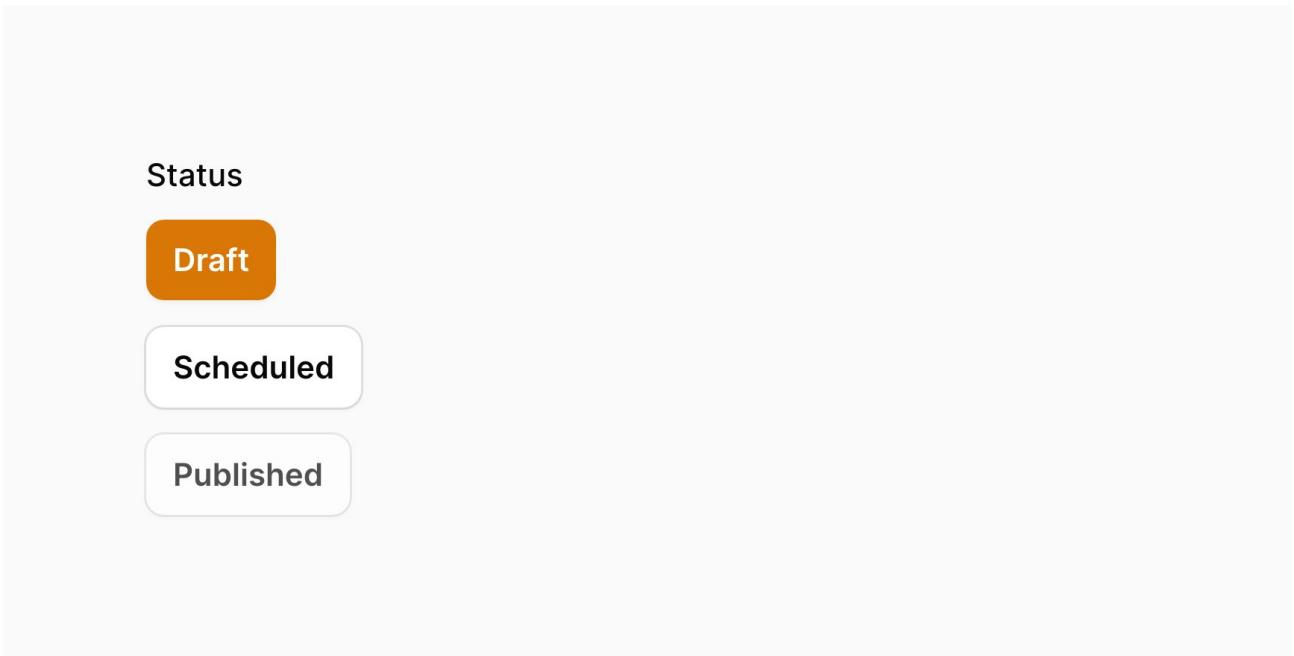


Disabling specific options

You can disable specific options using the `disableOptionWhen()` method. It accepts a closure, in which you can check if the option with a specific `$value` should be disabled:

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('status')
    ->options([
        'draft' => 'Draft',
        'scheduled' => 'Scheduled',
        'published' => 'Published',
    ])
    ->disableOptionWhen(fn (string $value): bool => $value === 'published')
```



If you want to retrieve the options that have not been disabled, e.g. for validation purposes, you can do so using

```
getEnabledOptions():
```

```
use Filament\Forms\Components\ToggleButtons;

ToggleButtons::make('status')
->options([
    'draft' => 'Draft',
    'scheduled' => 'Scheduled',
    'published' => 'Published',
])
->disableOptionWhen(fn (string $value): bool => $value === 'published')
->in(fn (ToggleButtons $component): array => array_keys($component->getEnabledOptions()))
```

Hidden

Overview

The hidden component allows you to create a hidden field in your form that holds a value.

```
use Filament\Forms\Components\Hidden;  
  
Hidden::make('token')
```

Please be aware that the value of this field is still editable by the user if they decide to use the browser's developer tools. You should not use this component to store sensitive or read-only information.

Custom

View fields

Aside from [building custom fields](#), you may create "view" fields which allow you to create custom fields without extra PHP classes.

```
use Filament\Forms\Components\ViewField;

ViewField::make('rating')
    ->view('filament.forms.components.range-slider')
```

This assumes that you have a `resources/views/filament/forms/components/range-slider.blade.php` file.

Passing data to view fields

You can pass a simple array of data to the view using `viewData()`:

```
use Filament\Forms\Components\ViewField;

ViewField::make('rating')
    ->view('filament.forms.components.range-slider')
    ->viewData([
        'min' => 1,
        'max' => 5,
    ])
```

However, more complex configuration can be achieved with a [custom field class](#).

Custom field classes

You may create your own custom field classes and views, which you can reuse across your project, and even release as a plugin to the community.

If you're just creating a simple custom field to use once, you could instead use a [view field](#) to render any custom Blade file.

To create a custom field class and view, you may use the following command:

```
php artisan make:form-field RangeSlider
```

This will create the following field class:

```
use Filament\Forms\Components\Field;

class RangeSlider extends Field
{
    protected string $view = 'filament.forms.components.range-slider';
}
```

It will also create a view file at `resources/views/filament/forms/components/range-slider.blade.php`.

How fields work

Livewire components are PHP classes that have their state stored in the user's browser. When a network request is made, the state is sent to the server, and filled into public properties on the Livewire component class, where it can be accessed in the same way as any other class property in PHP can be.

Imagine you had a Livewire component with a public property called `$name`. You could bind that property to an input field in the HTML of the Livewire component in one of two ways: with the `wire:model` attribute, or by entangling it with an Alpine.js property:

```
<input wire:model="name" />

<!-- Or -->

<div x-data="{ state: $wire.$entangle('name') }">
    <input x-model="state" />
</div>
```

When the user types into the input field, the `$name` property is updated in the Livewire component class. When the user submits the form, the `$name` property is sent to the server, where it can be saved.

This is the basis of how fields work in Filament. Each field is assigned to a public property in the Livewire component class, which is where the state of the field is stored. We call the name of this property the "state path" of the field. You can access the state path of a field using the `$getStatePath()` function in the field's view:

```
<input wire:model="{!! $getStatePath() !!}" />

<!-- Or -->

<div x-data="{ state: $wire.$entangle('{!! $getStatePath() !!}') }">
    <input x-model="state" />
</div>
```

If your component heavily relies on third party libraries, we advise that you asynchronously load the Alpine.js component using the Filament asset system. This ensures that the Alpine.js component is only loaded when it's needed, and not on every page load. To find out how to do this, check out our [Assets documentation](#).

Rendering the field wrapper

Filament includes a "field wrapper" component, which is able to render the field's label, validation errors, and any other text surrounding the field. You may render the field wrapper like this in the view:

```
<x-dynamic-component
    :component="$getFieldWrapperView()"
    :field="$field"
>
    <!-- Field -->
</x-dynamic-component>
```

It's encouraged to use the field wrapper component whenever appropriate, as it will ensure that the field's design is consistent with the rest of the form.

Accessing the Eloquent record

Inside your view, you may access the Eloquent record using the `$getRecord()` function:

```
<div>
  {{ $getRecord() ->name  }}
</div>
```

Obeying state binding modifiers

When you bind a field to a state path, you may use the `defer` modifier to ensure that the state is only sent to the server when the user submits the form, or whenever the next Livewire request is made. This is the default behavior.

However, you may use the `live()` on a field to ensure that the state is sent to the server immediately when the user interacts with the field. This allows for lots of advanced use cases as explained in the [advanced](#) section of the documentation.

Filament provides a `$applyStateBindingModifiers()` function that you may use in your view to apply any state binding modifiers to a `wire:model` or `$wire.$entangle()` binding:

```
<input {{ $applyStateBindingModifiers('wire:model') }}="{{ $getStatePath() }}" />

<!-- Or -->

<div x-data="{ state: $wire.{{ $applyStateBindingModifiers("\$entangle('{$getStatePath()}')") }} }">
  <input x-model="state" />
</div>
```

Layout

Getting Started

Overview

Filament forms are not limited to just displaying fields. You can also use "layout components" to organize them into an infinitely nestable structure.

Layout component classes can be found in the `Filament\Forms\Components` namespace. They reside within the schema of your form, alongside any [fields](#).

Components may be created using the static `make()` method. Usually, you will then define the child component `schema()` to display inside:

```
use Filament\Forms\Components\Grid;

Grid::make(2
    ->schema([
        // ...
    ])
```

Available layout components

Filament ships with some layout components, suitable for arranging your form fields depending on your needs:

- [Grid](#)
- [Fieldset](#)
- [Tabs](#)
- [Wizard](#)
- [Section](#)
- [Split](#)
- [Placeholder](#)

You may also [create your own custom layout components](#) to organize fields however you wish.

Setting an ID

You may define an ID for the component using the `id()` method:

```
use Filament\Forms\Components\Section;

Section::make()
    ->id('main-section')
```

Adding extra HTML attributes

You can pass extra HTML attributes to the component, which will be merged onto the outer DOM element. Pass an array of attributes to the `extraAttributes()` method, where the key is the attribute name and the value is the attribute value:

```
use Filament\Forms\Components\Section;

Section::make()
->extraAttributes(['class' => 'custom-section-style'])
```

Classes will be merged with the default classes, and any other attributes will override the default attributes.

Global settings

If you wish to change the default behavior of a component globally, then you can call the static `configureUsing()` method inside a service provider's `boot()` method, to which you pass a Closure to modify the component using. For example, if you wish to make all section components have [2 columns](#) by default, you can do it like so:

```
use Filament\Forms\Components\Section;

Section::configureUsing(function (Section $section): void {
    $section
        ->columns(2);
});
```

Of course, you are still able to overwrite this on each field individually:

```
use Filament\Forms\Components\Section;

Section::make()
->columns(1)
```

Grid

Overview

Filament's grid system allows you to create responsive, multi-column layouts using any layout component.

Responsively setting the number of grid columns

All layout components have a `columns()` method that you can use in a couple of different ways:

- You can pass an integer like `columns(2)`. This integer is the number of columns used on the `lg` breakpoint and higher. All smaller devices will have just 1 column.
- You can pass an array, where the key is the breakpoint and the value is the number of columns. For example, `columns(['md' => 2, 'xl' => 4])` will create a 2 column layout on medium devices, and a 4 column layout on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

Breakpoints (`sm`, `md`, `lg`, `xl`, `2xl`) are defined by Tailwind, and can be found in the [Tailwind documentation](#).

Controlling how many columns a component should span

In addition to specifying how many columns a layout component should have, you may also specify how many columns a component should fill within the parent grid, using the `columnSpan()` method. This method accepts an integer or an array of breakpoints and column spans:

- `columnSpan(2)` will make the component fill up to 2 columns on all breakpoints.
- `columnSpan(['md' => 2, 'xl' => 4])` will make the component fill up to 2 columns on medium devices, and up to 4 columns on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.
- `columnSpan('full')` or `columnSpanFull()` or `columnSpan(['default' => 'full'])` will make the component fill the full width of the parent grid, regardless of how many columns it has.

An example of a responsive grid layout

In this example, we have a form with a `section` layout component. Since all layout components support the `columns()` method, we can use it to create a responsive grid layout within the section itself.

We pass an array to `columns()` as we want to specify different numbers of columns for different breakpoints. On devices smaller than the `sm` [Tailwind breakpoint](#), we want to have 1 column, which is default. On devices larger than the `sm` breakpoint, we want to have 3 columns. On devices larger than the `xl` breakpoint, we want to have 6 columns. On devices larger than the `2xl` breakpoint, we want to have 8 columns.

Inside the section, we have a [text input](#). Since text inputs are form components and all form components have a `columnSpan()` method, we can use it to specify how many columns the text input should fill. On devices smaller than the `sm` breakpoint, we want the text input to fill 1 column, which is default. On devices larger than the `sm` breakpoint, we want the text input to fill 2 columns. On devices larger than the `xl` breakpoint, we want the text input to fill 3 columns. On devices larger than the `2xl` breakpoint, we want the text input to fill 4 columns.

```
use Filament\Forms\Components\Section;
use Filament\Forms\Components\TextInput;

Section::make()
->columns([
    'sm' => 3,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    TextInput::make('name')
        ->columnSpan([
            'sm' => 2,
            'xl' => 3,
            '2xl' => 4,
        ]),
    // ...
])
```

Grid component

All layout components support the `columns()` method, but you also have access to an additional `Grid` component. If you feel that your form schema would benefit from an explicit grid syntax with no extra styling, it may be useful to you. Instead of using the `columns()` method, you can pass your column configuration directly to `Grid::make()`:

```
use Filament\Forms\Components\Grid;

Grid::make([
    'default' => 1,
    'sm' => 2,
    'md' => 3,
    'lg' => 4,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    // ...
])
```

Setting the starting column of a component in a grid

If you want to start a component in a grid at a specific column, you can use the `columnStart()` method. This method accepts an integer, or an array of breakpoints and which column the component should start at:

- `columnStart(2)` will make the component start at column 2 on all breakpoints.
- `columnStart(['md' => 2, 'xl' => 4])` will make the component start at column 2 on medium devices, and at column 4 on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

```
use Filament\Forms\Components\Section;

Section::make()
->columns([
    'sm' => 3,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    TextInput::make('name')
        ->columnStart([
            'sm' => 2,
            'xl' => 3,
            '2xl' => 4,
        ]),
    // ...
])
```

In this example, the grid has 3 columns on small devices, 6 columns on extra large devices, and 8 columns on extra extra large devices. The text input will start at column 2 on small devices, column 3 on extra large devices, and column 4 on extra extra large devices. This is essentially producing a layout whereby the text input always starts halfway through the grid, regardless of how many columns the grid has.

Fieldset

Overview

You may want to group fields into a Fieldset. Each fieldset has a label, a border, and a two-column grid by default:

```
use Filament\Forms\Components\Fieldset;

Fieldset::make('Label')
    ->schema([
        // ...
    ])
```

The screenshot shows a 'Rate limiting' fieldset with three inputs: 'Hits' (30), 'Period' (Hour), and 'Maximum' (100).

Using grid columns within a fieldset

You may use the `columns()` method to customize the `grid` within the fieldset:

```
use Filament\Forms\Components\Fieldset;

Fieldset::make('Label')
    ->schema([
        // ...
    ])
    ->columns(3)
```

Tabs

Overview

Some forms can be long and complex. You may want to use tabs to reduce the number of components that are visible at once:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        Tabs\Tab::make('Tab 1')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 2')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 3')
            ->schema([
                // ...
            ]),
    ])
)
```

The screenshot shows a user interface with a top navigation bar containing three tabs: "Rate Limiting" (highlighted in orange), "Proxy", and "Meta". Below the tabs, there are three input fields arranged horizontally: "Hits" (with value "30"), "Period" (with dropdown menu showing "Hour"), and "Maximum" (with value "100"). Below these fields is a section labeled "Notes" with a text input area.

Setting the default active tab

The first tab will be open by default. You can change the default open tab using the `activeTab()` method:

```
use Filament\Forms\Components\tabs;

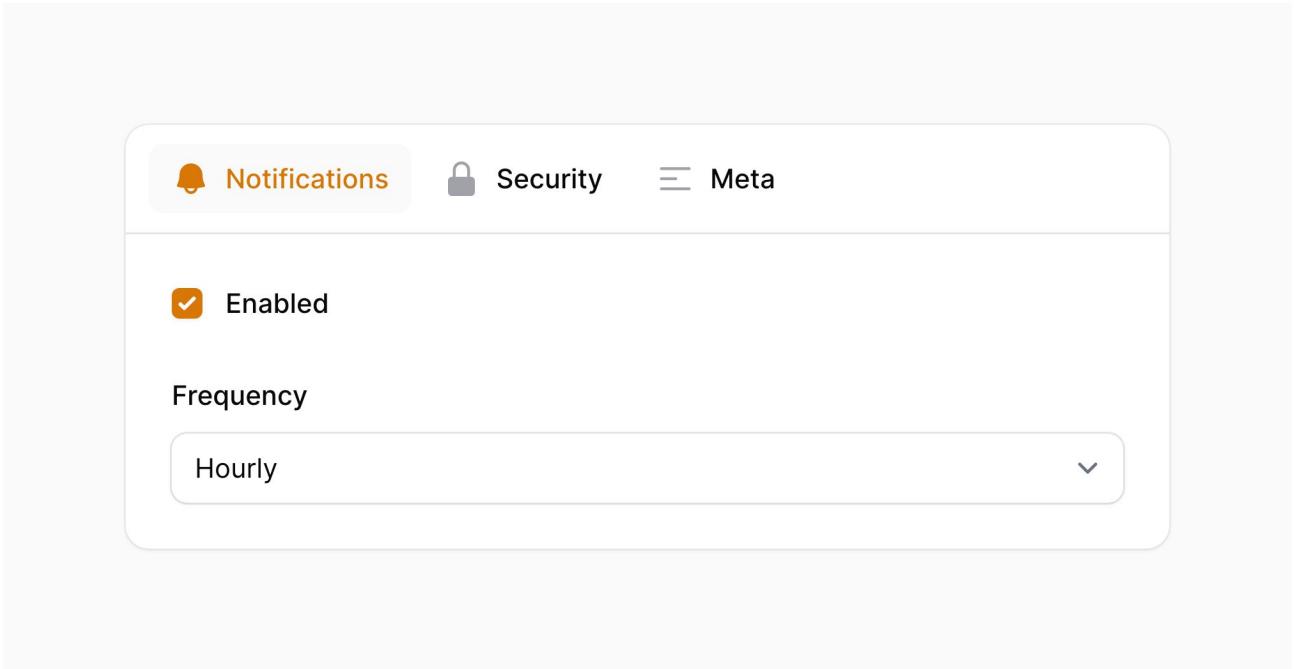
Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->activeTab(2)
```

Setting a tab icon

Tabs may have an `icon`, which you can set using the `icon()` method:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-m-bell')
        ->schema([
            // ...
        ]),
    // ...
])
```

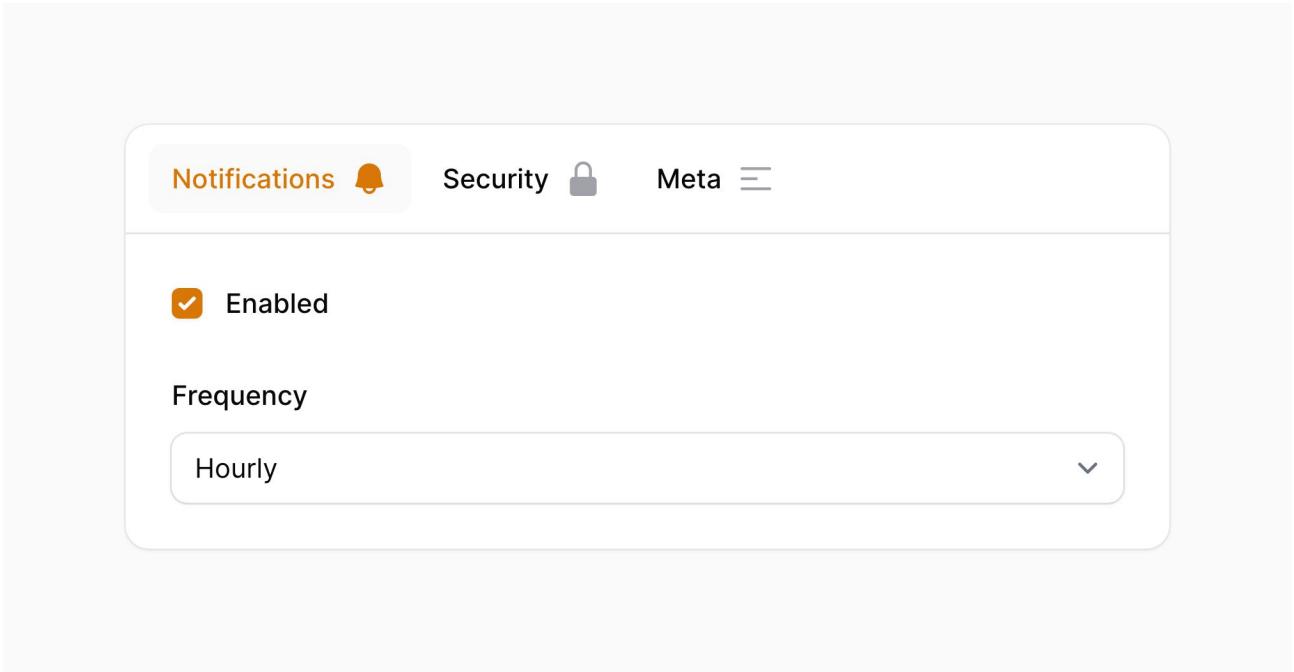


Setting the tab icon position

The icon of the tab may be positioned before or after the label using the `iconPosition()` method:

```
use Filament\Forms\Components\tabs;
use Filament\Support\Enums\IconPosition;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-m-bell')
        ->iconPosition(IconPosition::After)
        ->schema([
            // ...
        ]),
        // ...
    ])
])
```

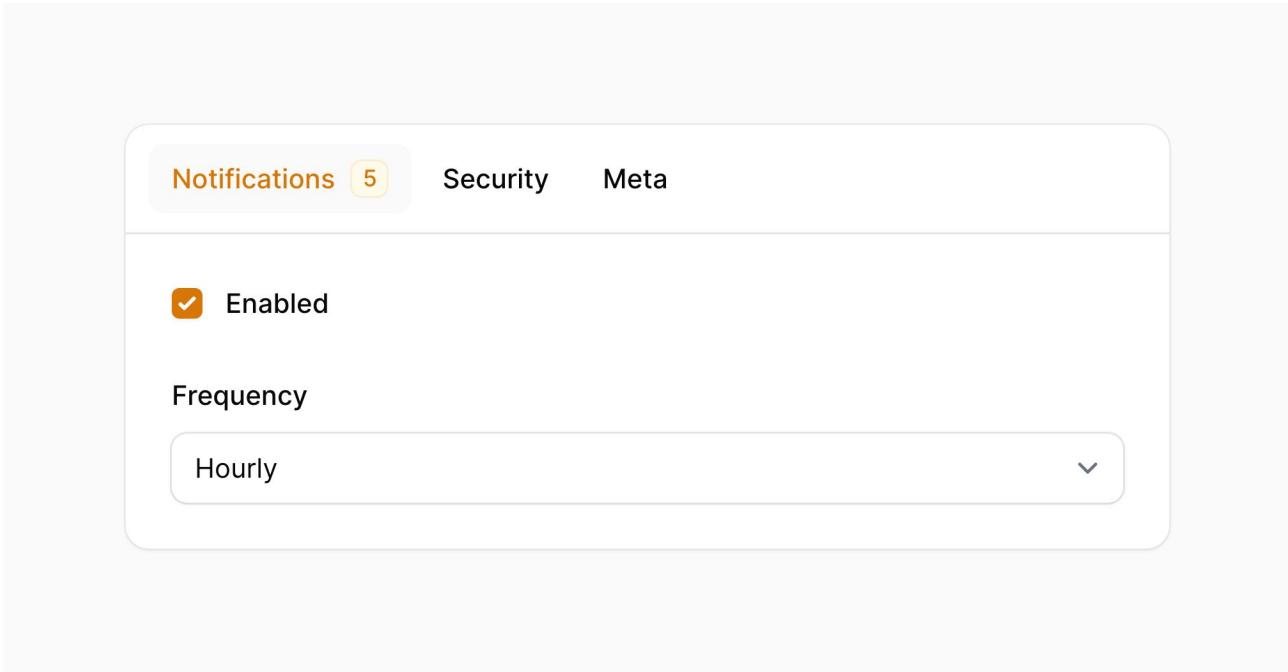


Setting a tab badge

Tabs may have a badge, which you can set using the `badge()` method:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->badge(5)
        ->schema([
            // ...
        ]),
        // ...
    ])
)
```



If you'd like to change the color for a badge, you can use the `badgeColor()` method:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        Tabs\Tab::make('Notifications')
            ->badge(5)
            ->badgeColor('success')
            ->schema([
                // ...
            ]),
            // ...
    ])
])
```

Using grid columns within a tab

You may use the `columns()` method to customize the `grid` within the tab:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        Tabs\Tab::make('Tab 1')
            ->schema([
                // ...
            ])
            ->columns(3),
            // ...
    ])
])
```

Removing the styled container

By default, tabs and their content are wrapped in a container styled as a card. You may remove the styled container using

`contained()`:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        Tabs\Tab::make('Tab 1')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 2')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 3')
            ->schema([
                // ...
            ]),
    ])
->contained(false)
```

Persisting the current tab

By default, the current tab is not persisted in the browser's local storage. You can change this behavior using the `persistTab()` method. You must also pass in a unique `id()` for the tabs component, to distinguish it from all other sets of tabs in the app. This ID will be used as the key in the local storage to store the current tab:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        // ...
    ])
->persistent()
->id('order-tabs')
```

Persisting the current tab in the URL's query string

By default, the current tab is not persisted in the URL's query string. You can change this behavior using the `persistTabInQueryString()` method:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->persistentTabInQueryString()
```

By default, the current tab is persisted in the URL's query string using the `tab` key. You can change this key by passing it to the `persistentTabInQueryString()` method:

```
use Filament\Forms\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->persistentTabInQueryString('settings-tab')
```

Wizard

Overview

Similar to [tabs](#), you may want to use a multistep form wizard to reduce the number of components that are visible at once. These are especially useful if your form has a definite chronological order, in which you want each step to be validated as the user progresses.

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    Wizard\Step::make('Order')
        ->schema([
            // ...
        ]),
    Wizard\Step::make('Delivery')
        ->schema([
            // ...
        ]),
    Wizard\Step::make('Billing')
        ->schema([
            // ...
        ]),
])
```

The screenshot shows a three-step wizard form. The first step, 'Order', is active and highlighted with an orange circle containing '01'. It contains fields for 'Product' (a dropdown menu showing 'Filament t-shirt') and 'Quantity' (a text input showing '3'). There is also a red trash icon in the top right corner of this section. Below these fields is a button labeled 'Add to order'. The second step, 'Delivery', is shown with a grey circle containing '02' and is currently inactive. The third step, 'Billing', is shown with a grey circle containing '03' and is also inactive. At the bottom right of the form is a large orange 'Next' button.

We have different setup instructions if you're looking to add a wizard to the creation process inside a [panel resource](#) or an [action modal](#). Following that documentation will ensure that the ability to submit the form is only available on the last step of the wizard.

Rendering a submit button on the last step

You may use the `submitAction()` method to render submit button HTML or a view at the end of the wizard, on the last step. This provides a clearer UX than displaying a submit button below the wizard at all times:

```
use Filament\Forms\Components\Wizard;
use Illuminate\Support\HtmlString;

Wizard::make([
    // ...
])->submitAction(view('order-form.submit-button'))

Wizard::make([
    // ...
])->submitAction(new HtmlString('<button type="submit">Submit</button>'))
```

Alternatively, you can use the built-in Filament button Blade component:

```
use Filament\Forms\Components\Wizard;
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\HtmlString;

Wizard::make([
    // ...
])->submitAction(new HtmlString(Blade::render(<<<BLADE
<x-filament::button
    type="submit"
    size="sm"
>
    Submit
</x-filament::button>
BLADE)))
```

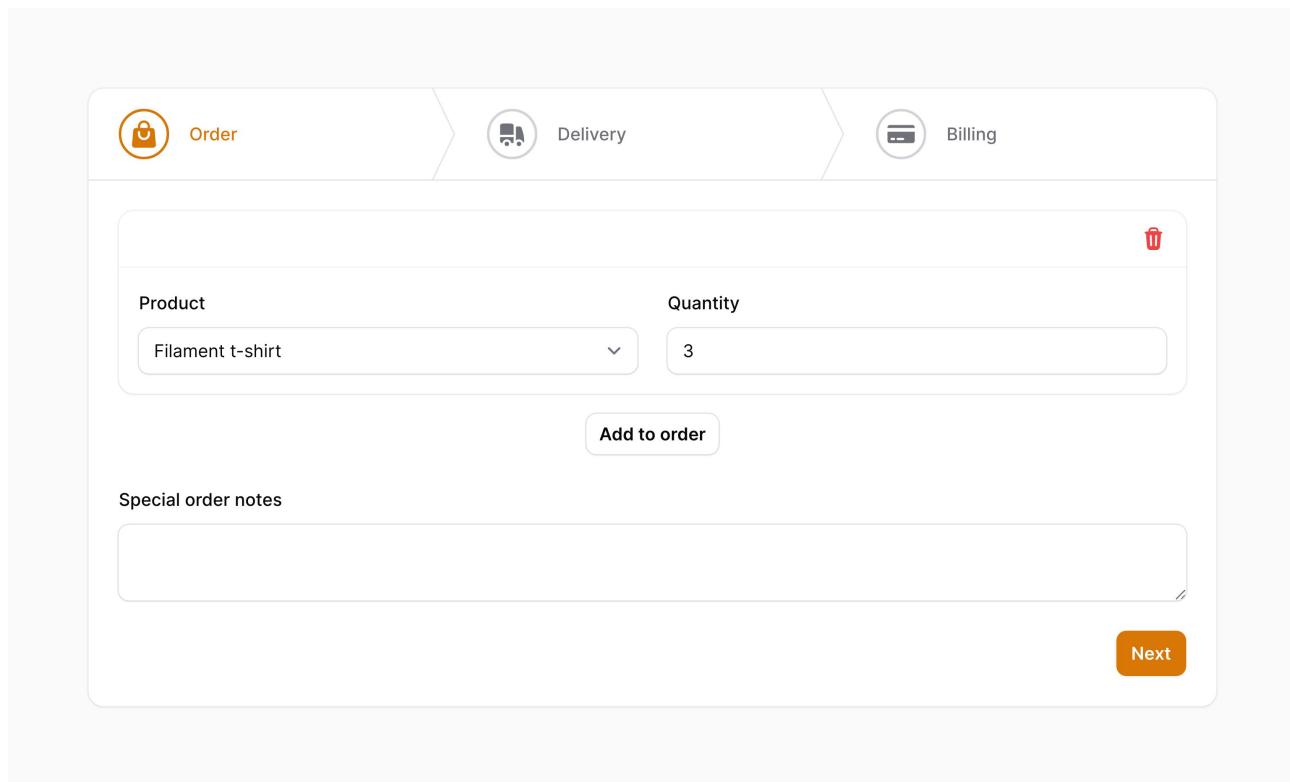
You could use this component in a separate Blade view if you want.

Setting up step icons

Steps may also have an `icon`, set using the `icon()` method:

```
use Filament\Forms\Components\Wizard;

Wizard\Step::make('Order')
    ->icon('heroicon-m-shopping-bag')
    ->schema([
        // ...
    ]),
```

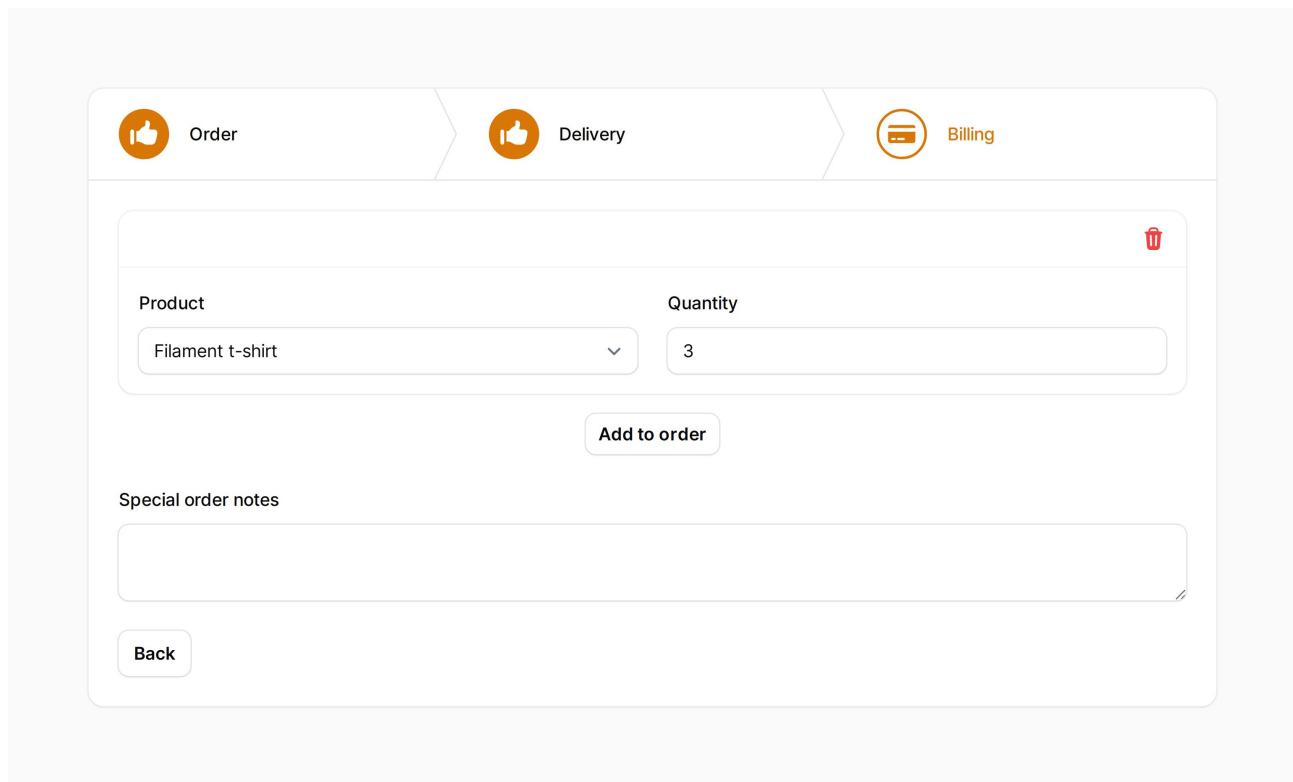


Customizing the icon for completed steps

You may customize the `icon` of a completed step using the `completedIcon()` method:

```
use Filament\Forms\Components\Wizard;

Wizard\Step::make('Order')
    ->completedIcon('heroicon-m-hand-thumb-up')
    ->schema([
        // ...
    ]),
```

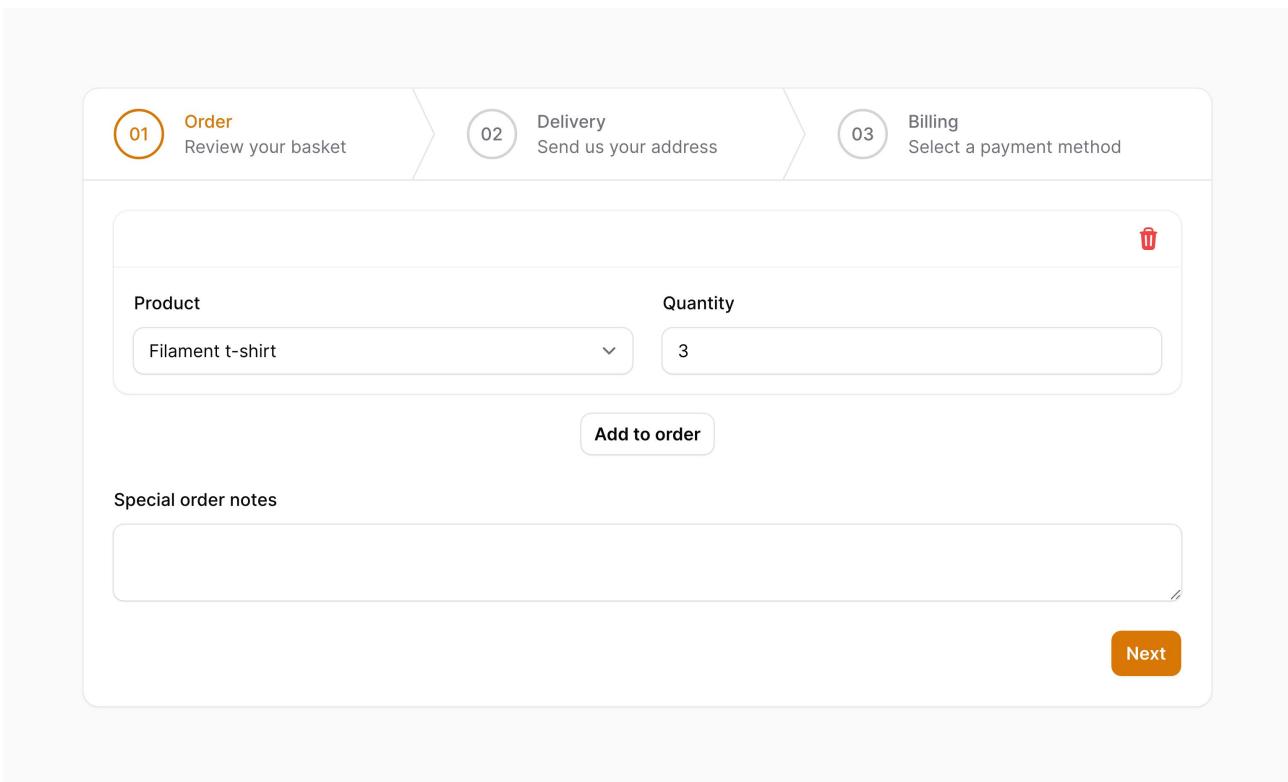


Adding descriptions to steps

You may add a short description after the title of each step using the `description()` method:

```
use Filament\Forms\Components\Wizard;

Wizard::Step::make('Order')
    ->description('Review your basket')
    ->schema([
        // ...
    ]),
```



Setting the default active step

You may use the `startOnStep()` method to load a specific step in the wizard:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])->startOnStep(2)
```

Allowing steps to be skipped

If you'd like to allow free navigation, so all steps are skippable, use the `skippable()` method:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])->skippable()
```

Persisting the current step in the URL's query string

By default, the current step is not persisted in the URL's query string. You can change this behavior using the `persistStepInQueryString()` method:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])->persistStepInQueryString()
```

By default, the current step is persisted in the URL's query string using the `step` key. You can change this key by passing it to the `persistStepInQueryString()` method:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])->persistStepInQueryString('wizard-step')
```

Step lifecycle hooks

You may use the `afterValidation()` and `beforeValidation()` methods to run code before and after validation occurs on the step:

```
use Filament\Forms\Components\Wizard;

Wizard\Step::make('Order')
    ->afterValidation(function () {
        // ...
    })
    ->beforeValidation(function () {
        // ...
    })
    ->schema([
        // ...
    ]),
```

Preventing the next step from being loaded

Inside `afterValidation()` or `beforeValidation()`, you may throw `Filament\Support\Exceptions\Halt`, which will prevent the wizard from loading the next step:

```
use Filament\Forms\Components\Wizard;
use Filament\Support\Exceptions\Halt;

Wizard\Step::make('Order')
    ->afterValidation(function () {
        // ...

        if (true) {
            throw new Halt();
        }
    })
    ->schema([
        // ...
    ]),
```

Using grid columns within a step

You may use the `columns()` method to customize the `grid` within the step:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    Wizard\Step::make('Order')
        ->columns(2)
        ->schema([
            // ...
        ]),
        // ...
])
```

Customizing the wizard action objects

This component uses action objects for easy customization of buttons within it. You can customize these buttons by passing a function to an action registration method. The function has access to the `$action` object, which you can use to customize it. The following methods are available to customize the actions:

- `nextAction()`
- `previousAction()`

Here is an example of how you might customize an action:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])

->nextAction(
    fn (Action $action) => $action->label('Next step'),
)
```

Section

Overview

You may want to separate your fields into sections, each with a heading and description. To do this, you can use a section component:

```
use Filament\Forms\Components\Section;

Section::make('Rate limiting')
    ->description('Prevent abuse by limiting the number of requests per period')
    ->schema([
        // ...
    ])
```

Hits	Period	Maximum
30	Hour	100

Notes

You can also use a section without a header, which just wraps the components in a simple card:

```
use Filament\Forms\Components\Section;

Section::make()
    ->schema([
        // ...
    ])
```

Hits	Period	Maximum
30	Hour	100

Notes

Adding actions to the section's header or footer

Sections can have actions in their header or footer.

Adding actions to the section's header

You may add actions to the section's header using the `headerActions()` method:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Section;

Section::make('Rate limiting')
    ->headerActions([
        Action::make('test')
            ->action(function () {
                // ...
            }),
    ])
    ->schema([
        // ...
    ])
```

Rate limiting

Test

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
<input type="text" value="30"/>	<input style="width: 100px;" type="text" value="Hour"/> ▼	<input type="text" value="100"/>

Notes

Make sure the section has a heading or ID

Adding actions to the section's footer

In addition to header actions, you may add actions to the section's footer using the `footerActions()` method:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Section;

Section::make('Rate limiting')
    ->schema([
        // ...
    ])
    ->footerActions([
        Action::make('test')
            ->action(function () {
                // ...
            }),
    ])
)
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour ▼	100
Notes		
Test		

Make sure the section has a heading or ID

Aligning section footer actions

Footer actions are aligned to the inline start by default. You may customize the alignment using the `footerActionsAlignment()` method:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\Section;
use Filament\Support\Enums\Alignment;

Section::make('Rate limiting')
    ->schema([
        // ...
    ])
    ->footerActions([
        Action::make('test')
            ->action(function () {
                // ...
            }),
    ])
    ->footerActionsAlignment(Alignment::End)
```

Adding actions to a section without heading

If your section does not have a heading, Filament has no way of locating the action inside it. In this case, you must pass a unique `id()` to the section:

```
use Filament\Forms\Components\Section;

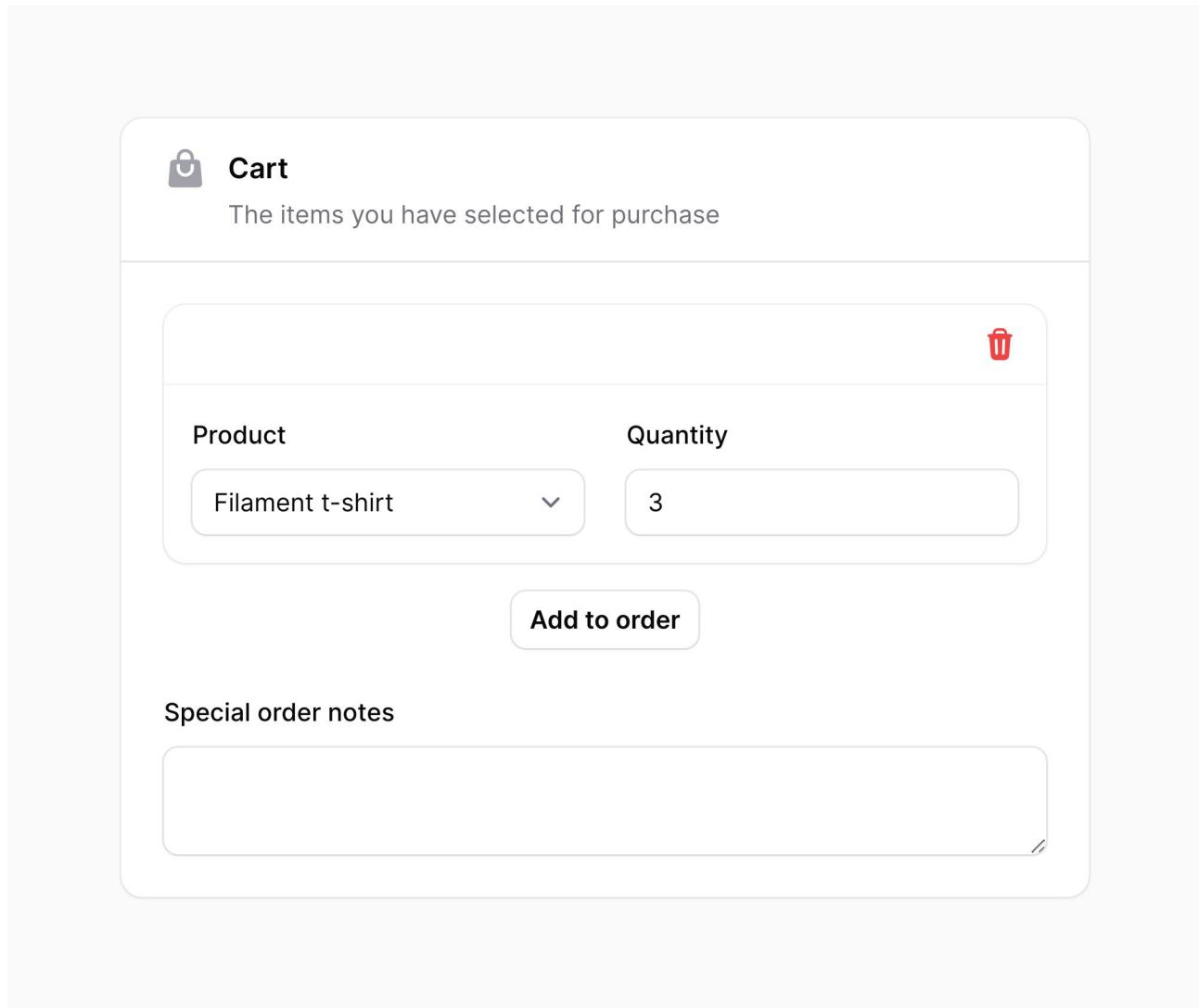
Section::make()
->id('rateLimitingSection')
->headerActions([
    // ...
])
->schema([
    // ...
])
```

Adding an icon to the section's header

You may add an `icon` to the section's header using the `icon()` method:

```
use Filament\Forms\Components\Section;

Section::make('Cart')
->description('The items you have selected for purchase')
->icon('heroicon-m-shopping-bag')
->schema([
    // ...
])
```



Positioning the heading and description aside

You may use the `aside()` to align heading & description on the left, and the form components inside a card on the right:

```
use Filament\Forms\Components\Section;

Section::make('Rate limiting')
    ->description('Prevent abuse by limiting the number of requests per period')
    ->aside([
        // ...
    ])
```

Rate limiting

Prevent abuse by limiting the number of requests per period

The screenshot shows a configuration panel for rate limiting. It has four main sections: 'Hits' (set to 30), 'Period' (set to 'Hour'), 'Maximum' (set to 100), and a large 'Notes' section which is currently empty.

Collapsing sections

Sections may be `collapsible()` to optionally hide content in long forms:

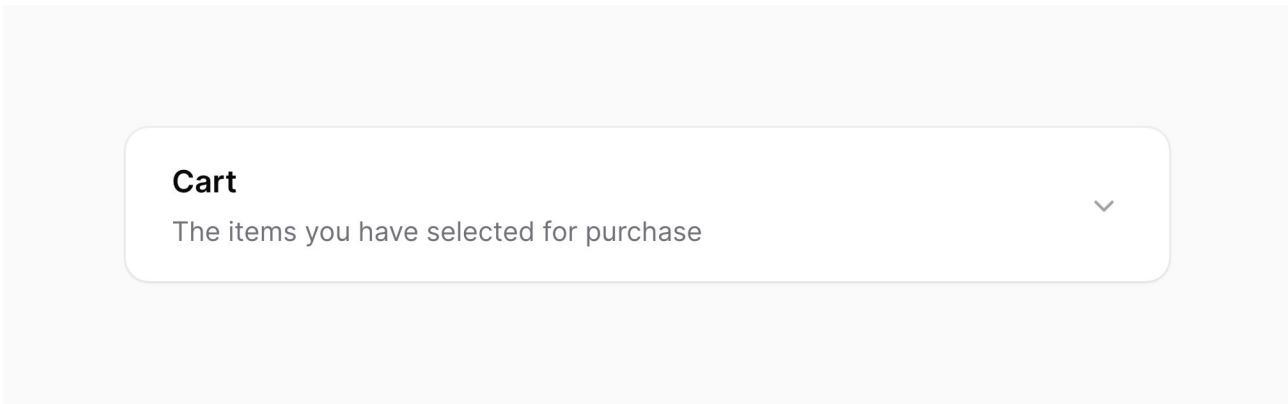
```
use Filament\Forms\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
```

Your sections may be `collapsed()` by default:

```
use Filament\Forms\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsed()
```



Persisting collapsed sections

You can persist whether a section is collapsed in local storage using the `persistCollapsed()` method, so it will remain collapsed when the user refreshes the page:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
    ->persistentCollapsed()
```

To persist the collapse state, the local storage needs a unique ID to store the state. This ID is generated based on the heading of the section. If your section does not have a heading, or if you have multiple sections with the same heading that you do not want to collapse together, you can manually specify the `id()` of that section to prevent an ID conflict:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
    ->persistentCollapsed()
    ->id('order-cart')
```

Compact section styling

When nesting sections, you can use a more compact styling:

```
use Filament\Forms\Components\Section;

Section::make('Rate limiting')
    ->description('Prevent abuse by limiting the number of requests per period')
    ->schema([
        // ...
    ])
    ->compact()
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour ▾	100

Notes

Using grid columns within a section

You may use the `columns()` method to easily create a grid within the section:

```
use Filament\Forms\Components\Section;

Section::make('Heading')
    ->schema([
        // ...
    ])
    ->columns(2)
```

Split

Overview

The `Split` component allows you to define layouts with flexible widths, using flexbox.

```
use Filament\Forms\Components\Section;
use Filament\Forms\Components\Split;
use Filament\Forms\Components\Textarea;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Toggle;

Split::make([
    Section::make([
        TextInput::make('title'),
        Textarea::make('content'),
    ]),
    Section::make([
        Toggle::make('is_published'),
        Toggle::make('is_featured'),
    ]) ->grow(false),
]) ->from('md')
```

In this example, the first section will `grow()` to consume available horizontal space, without affecting the amount of space needed to render the second section. This creates a sidebar effect.

The `from()` method is used to control the Tailwind breakpoint (`sm`, `md`, `lg`, `xl`, `2xl`) at which the split layout should be used. In this example, the split layout will be used on medium devices and larger. On smaller devices, the sections will stack on top of each other.

Title

 Lorem ipsum dolor sit amet

Content

 Lorem ipsum dolor sit amet,
 consectetur adipiscing elit. Donec
 euismod, nisl eget tempor aliquam,
 nunc nisl aliquet nunc, quis aliquam nisl
 nunc quis nisl. Donec euismod, nisl



Is published



Is featured

Custom

View components

Aside from [building custom layout components](#), you may create "view" components which allow you to create custom layouts without extra PHP classes.

```
use Filament\Forms\Components\View;

View::make('filament.forms.components.wizard')
```

This assumes that you have a `resources/views/filament/forms/components/wizard.blade.php` file.

Custom layout classes

You may create your own custom component classes and views, which you can reuse across your project, and even release as a plugin to the community.

If you're just creating a simple custom component to use once, you could instead use a [view component](#) to render any custom Blade file.

To create a custom column class and view, you may use the following command:

```
php artisan make:form-layout Wizard
```

This will create the following layout component class:

```
use Filament\Forms\Components\Component;

class Wizard extends Component
{
    protected string $view = 'filament.forms.components.wizard';

    public static function make(): static
    {
        return app(static::class);
    }
}
```

It will also create a view file at `resources/views/filament/forms/components/wizard.blade.php`.

Rendering the component's schema

Inside your view, you may render the component's `schema()` using the `$getChildComponentContainer()` function:

```
<div>
    {{ $getChildComponentContainer() }}
</div>
```

Accessing the Eloquent record

Inside your view, you may access the Eloquent record using the `$getRecord()` function:

```
<div>
    {{ $getRecord() ->name  } }
</div>
```

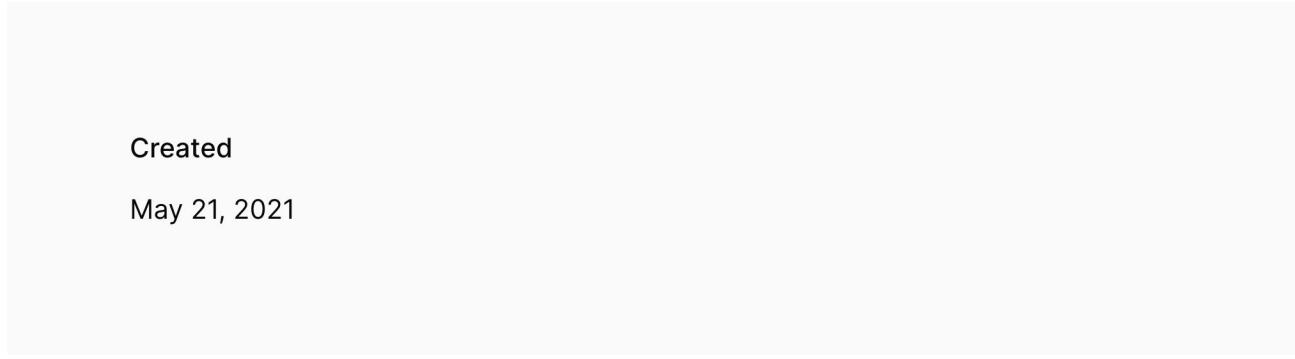
Placeholder

Overview

Placeholders can be used to render text-only "fields" within your forms. Each placeholder has `content()`, which cannot be changed by the user.

```
use App\Models\Post;
use Filament\Forms\Components\Placeholder;

Placeholder::make('created')
->content(fn (Post $record): string => $record->created_at->toFormattedDateString())
```



Created

May 21, 2021

Important: All form fields require a unique name. That also applies to Placeholders!

Rendering HTML inside the placeholder

You may even render custom HTML within placeholder content:

```
use Filament\Forms\Components\Placeholder;
use Illuminate\Support\HtmlString;

Placeholder::make('documentation')
->content(new HtmlString('<a href="https://filamentphp.com/docs">filamentphp.com</a>'))
```

Dynamically generating placeholder content

By passing a closure to the `content()` method, you may dynamically generate placeholder content. You have access to any closure parameter explained in the [advanced closure customization](#) documentation:

```
use Filament\Forms\Components\Placeholder;
use Filament\Forms\Get;

Placeholder::make('total')
->content(function (Get $get): string {
    return '€' . number_format($get('cost') * $get('quantity'), 2);
})
```

Validation

Overview

Validation rules may be added to any [field](#).

In Laravel, validation rules are usually defined in arrays like `['required', 'max:255']` or a combined string like `required|max:255`. This is fine if you're exclusively working in the backend with simple form requests. But Filament is also able to give your users frontend validation, so they can fix their mistakes before any backend requests are made.

Filament includes several [dedicated validation methods](#), but you can also use any [other Laravel validation rules](#), including [custom validation rules](#).

Beware that some validations rely on the field name and therefore won't work when passed via `->rule()` / `->rules()`. Use the dedicated validation methods whenever you can.

Available rules

Active URL

The field must have a valid A or AAAA record according to the `dns_get_record()` PHP function. [See the Laravel documentation](#).

```
Field::make('name')->activeUrl()
```

After (date)

The field value must be a value after a given date. [See the Laravel documentation](#).

```
Field::make('start_date')->after('tomorrow')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('start_date')
Field::make('end_date')->after('start_date')
```

After or equal to (date)

The field value must be a date after or equal to the given date. [See the Laravel documentation](#).

```
Field::make('start_date')->afterOrEqual('tomorrow')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('start_date')
Field::make('end_date')->afterOrEqual('start_date')
```

Alpha

The field must be entirely alphabetic characters. [See the Laravel documentation](#).

```
Field::make('name')->alpha()
```

Alpha Dash

The field may have alphanumeric characters, as well as dashes and underscores. [See the Laravel documentation.](#)

```
Field::make('name')->alphaDash()
```

Alpha Numeric

The field must be entirely alphanumeric characters. [See the Laravel documentation.](#)

```
Field::make('name')->alphaNum()
```

ASCII

The field must be entirely 7-bit ASCII characters. [See the Laravel documentation.](#)

```
Field::make('name')->ascii()
```

Before (date)

The field value must be a date before a given date. [See the Laravel documentation.](#)

```
Field::make('start_date')->before('first day of next month')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('start_date')->before('end_date')
Field::make('end_date')
```

Before or equal to (date)

The field value must be a date before or equal to the given date. [See the Laravel documentation.](#)

```
Field::make('start_date')->beforeOrEqual('end of this month')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('start_date')->beforeOrEqual('end_date')
Field::make('end_date')
```

Confirmed

The field must have a matching field of `{field}_confirmation`. [See the Laravel documentation.](#)

```
Field::make('password')->confirmed()
Field::make('password_confirmation')
```

Different

The field value must be different to another. [See the Laravel documentation.](#)

```
Field::make('backup_email')->different('email')
```

Doesnt Start With

The field must not start with one of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->doesntStartWith(['admin'])
```

Doesnt End With

The field must not end with one of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->doesntEndWith(['admin'])
```

Ends With

The field must end with one of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->endsWith(['bot'])
```

Enum

The field must contain a valid enum value. [See the Laravel documentation.](#)

```
Field::make('status')->enum(MyStatus::class)
```

Exists

The field value must exist in the database. [See the Laravel documentation.](#)

```
Field::make('invitation')->exists()
```

By default, the form's model will be searched, [if it is registered](#). You may specify a custom table name or model to search:

```
use App\Models\Invitation;

Field::make('invitation')->exists(table: Invitation::class)
```

By default, the field name will be used as the column to search. You may specify a custom column to search:

```
Field::make('invitation')->exists(column: 'id')
```

You can further customize the rule by passing a [closure](#) to the `callback` parameter:

```
use Illuminate\Validation\Rules\Exists;

Field::make('invitation')
->exists(modifyRuleUsing: function (Exists $rule) {
    return $rule->where('is_active', 1);
})
```

Filled

The field must not be empty when it is present. [See the Laravel documentation.](#)

```
Field::make('name')->filled()
```

Greater than

The field value must be greater than another. [See the Laravel documentation.](#)

```
Field::make('newNumber')->gt('oldNumber')
```

Greater than or equal to

The field value must be greater than or equal to another. [See the Laravel documentation.](#)

```
Field::make('newNumber')->gte('oldNumber')
```

Hex color

The field value must be a valid color in hexadecimal format. [See the Laravel documentation.](#)

```
Field::make('color')->hexColor()
```

In

The field must be included in the given list of values. [See the Laravel documentation.](#)

```
Field::make('status')->in(['pending', 'completed'])
```

IP Address

The field must be an IP address. [See the Laravel documentation.](#)

```
Field::make('ip_address')->ip()
Field::make('ip_address')->ipv4()
Field::make('ip_address')->ipv6()
```

JSON

The field must be a valid JSON string. [See the Laravel documentation.](#)

```
Field::make('ip_address')->json()
```

Less than

The field value must be less than another. [See the Laravel documentation.](#)

```
Field::make('newNumber')->lt('oldNumber')
```

Less than or equal to

The field value must be less than or equal to another. [See the Laravel documentation.](#)

```
Field::make('newNumber')->lte('oldNumber')
```

Mac Address

The field must be a MAC address. [See the Laravel documentation.](#)

```
Field::make('mac_address')->macAddress()
```

Multiple Of

The field must be a multiple of value. [See the Laravel documentation.](#)

```
Field::make('number')->multipleOf(2)
```

Not In

The field must not be included in the given list of values. [See the Laravel documentation.](#)

```
Field::make('status')->notIn(['cancelled', 'rejected'])
```

Not Regex

The field must not match the given regular expression. [See the Laravel documentation.](#)

```
Field::make('email')->notRegex('/^.+$/i')
```

Nullable

The field value can be empty. This rule is applied by default if the `required` rule is not present. [See the Laravel documentation.](#)

```
Field::make('name')->nullable()
```

Prohibited

The field value must be empty. [See the Laravel documentation.](#)

```
Field::make('name')->prohibited()
```

Prohibited If

The field must be empty *only if* the other specified field has any of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->prohibitedIf('field', 'value')
```

Prohibited Unless

The field must be empty *unless* the other specified field has any of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->prohibitedUnless('field', 'value')
```

Prohibits

If the field is not empty, all other specified fields must be empty. [See the Laravel documentation.](#)

```
Field::make('name')->prohibits('field')
```

```
Field::make('name')->prohibits(['field', 'another_field'])
```

Required

The field value must not be empty. [See the Laravel documentation.](#)

```
Field::make('name')->required()
```

Required If

The field value must not be empty *only if* the other specified field has any of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->requiredIf('field', 'value')
```

Required If Accepted

The field value must not be empty *only if* the other specified field is equal to "yes", "on", 1, "1", true, or "true". [See the Laravel documentation.](#)

```
Field::make('name')->requiredIfAccepted('field')
```

Required Unless

The field value must not be empty *unless* the other specified field has any of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->requiredUnless('field', 'value')
```

Required With

The field value must not be empty *only if* any of the other specified fields are not empty. [See the Laravel documentation.](#)

```
Field::make('name')->requiredWith('field,another_field')
```

Required With All

The field value must not be empty *only if* all the other specified fields are not empty. [See the Laravel documentation.](#)

```
Field::make('name')->requiredWithAll('field,another_field')
```

Required Without

The field value must not be empty *only when* any of the other specified fields are empty. [See the Laravel documentation.](#)

```
Field::make('name')->requiredWithout('field,another_field')
```

Required Without All

The field value must not be empty *only when* all the other specified fields are empty. [See the Laravel documentation.](#)

```
Field::make('name')->requiredWithoutAll('field,another_field')
```

Regex

The field must match the given regular expression. [See the Laravel documentation.](#)

```
Field::make('email')->regex('/^.+@.+\$/i')
```

Same

The field value must be the same as another. [See the Laravel documentation.](#)

```
Field::make('password')->same('passwordConfirmation')
```

Starts With

The field must start with one of the given values. [See the Laravel documentation.](#)

```
Field::make('name')->startsWith(['a'])
```

String

The field must be a string. [See the Laravel documentation.](#)

```
Field::make('name')->string()
```

Unique

The field value must not exist in the database. [See the Laravel documentation.](#)

```
Field::make('email')->unique()
```

By default, the form's model will be searched, if it is registered. You may specify a custom table name or model to search:

```
use App\Models\User;

Field::make('email')->unique(table: User::class)
```

By default, the field name will be used as the column to search. You may specify a custom column to search:

```
Field::make('email')->unique(column: 'email_address')
```

Sometimes, you may wish to ignore a given model during unique validation. For example, consider an "update profile" form that includes the user's name, email address, and location. You will probably want to verify that the email address is unique. However, if the user only changes the name field and not the email field, you do not want a validation error to be thrown because the user is already the owner of the email address in question.

```
Field::make('email')->unique(ignoreable: $ignoredUser)
```

If you're using the [Panel Builder](#), you can easily ignore the current record by using `ignoreRecord` instead:

```
Field::make('email')->unique(ignoreRecord: true)
```

You can further customize the rule by passing a [closure](#) to the `modifyRuleUsing` parameter:

```
use Illuminate\Validation\Rules\Unique;

Field::make('email')
    ->unique(modifyRuleUsing: function (Unique $rule) {
        return $rule->where('is_active', 1);
    })
}
```

ULID

The field under validation must be a valid [Universally Unique Lexicographically Sortable Identifier](#) (ULID). [See the Laravel documentation.](#)

```
Field::make('identifier')->ulid()
```

UUID

The field must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID). [See the Laravel documentation.](#)

```
Field::make('identifier')->uuid()
```

Other rules

You may add other validation rules to any field using the `rules()` method:

```
TextInput::make('slug')->rules(['alpha_dash'])
```

A full list of validation rules may be found in the [Laravel documentation](#).

Custom rules

You may use any custom validation rules as you would do in [Laravel](#):

```
TextInput::make('slug')->rules([new Uppercase()])
```

You may also use [closure rules](#):

```
use Closure;

TextInput::make('slug')->rules([
    fn (): Closure => function (string $attribute, $value, Closure $fail) {
        if ($value === 'foo') {
            $fail('The :attribute is invalid.');
        }
    },
])
])
```

You may inject utilities like `$get` into your custom rules, for example if you need to reference other field values in your form:

```
use Closure;
use Filament\Forms\Get;

TextInput::make('slug')->rules([
    fn (Get $get): Closure => function (string $attribute, $value, Closure $fail) use ($get) {
        if ($get('other_field') === 'foo' && $value !== 'bar') {
            $fail("The {$attribute} is invalid.");
        }
    },
])
])
```

Customizing validation attributes

When fields fail validation, their label is used in the error message. To customize the label used in field error messages, use the `validationAttribute()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->validationAttribute('full name')
```

Validation messages

By default Laravel's validation error message is used. To customize the error messages, use the `validationMessages()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('email')
    ->unique("// ...")
    ->validationMessages([
        'unique' => 'The :attribute has already been registered.',
    ])
])
```

Sending validation notifications

If you want to send a notification when validation error occurs, you may do so by using the `onValidationError()` method on your Livewire component:

```
use Filament\Notifications\Notification;
use Illuminate\Validation\ValidationException;

protected function onValidationError(ValidationException $exception): void
{
    Notification::make()
        ->title($exception->getMessage())
        ->danger()
        ->send();
}
```

Alternatively, if you are using the Panel Builder and want this behavior on all the pages, add this inside the `boot()` method of your `AppServiceProvider`:

```
use Filament\Notifications\Notification;
use Filament\Pages\Page;
use Illuminate\Validation\ValidationException;

Page::$reportValidationErrorUsing = function (ValidationException $exception) {
    Notification::make()
        ->title($exception->getMessage())
        ->danger()
        ->send();
};
```

Actions

Overview

Filament's forms can use [Actions](#). They are buttons that can be added to any form component. For instance, you may want an action to call an API endpoint to generate content with AI, or to create a new option for a select dropdown. Also, you can [render anonymous sets of actions](#) on their own which are not attached to a particular form component.

Defining a form component action

Action objects inside a form component are instances of [Filament/Forms/Components/Actions/Action](#). You must pass a unique name to the action's `make()` method, which is used to identify it amongst others internally within Filament. You can [customize the trigger button](#) of an action, and even [open a modal](#) with little effort:

```
use App\Actions\ResetStars;
use Filament\Forms\Components\Actions\Action;

Action::make('resetStars')
    ->icon('heroicon-m-x-mark')
    ->color('danger')
    ->requiresConfirmation()
    ->action(function (ResetStars $resetStars) {
        $resetStars();
    })
}
```

Adding an affix action to a field

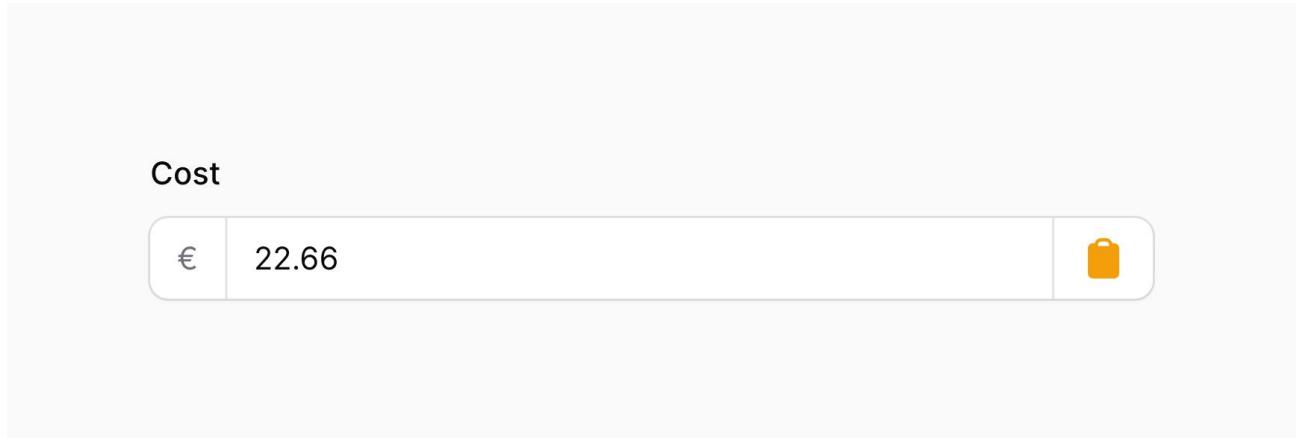
Certain fields support "affix actions", which are buttons that can be placed before or after its input area. The following fields support affix actions:

- [Text input](#)
- [Select](#)
- [Date-time picker](#)
- [Color picker](#)

To define an affix action, you can pass it to either `prefixAction()` or `suffixAction()`:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Set;

TextInput::make('cost')
    ->prefix('€')
    ->suffixAction(
        Action::make('copyCostToPrice')
            ->icon('heroicon-m-clipboard')
            ->requiresConfirmation()
            ->action(function (Set $set, $state) {
                $set('price', $state);
            })
    )
}
```



Notice `$set` and `$state` injected into the `action()` function in this example. This is [form component action utility injection](#).

Passing multiple affix actions to a field

You may pass multiple affix actions to a field by passing them in an array to either `prefixActions()` or `suffixActions()`. Either method can be used, or both at once, Filament will render all the registered actions in order:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\TextInput;

TextInput::make('cost')
    ->prefix('€')
    ->prefixActions([
        Action::make('...'),
        Action::make('...'),
        Action::make('...'),
    ])
    ->suffixActions([
        Action::make('...'),
        Action::make('...'),
    ])
```

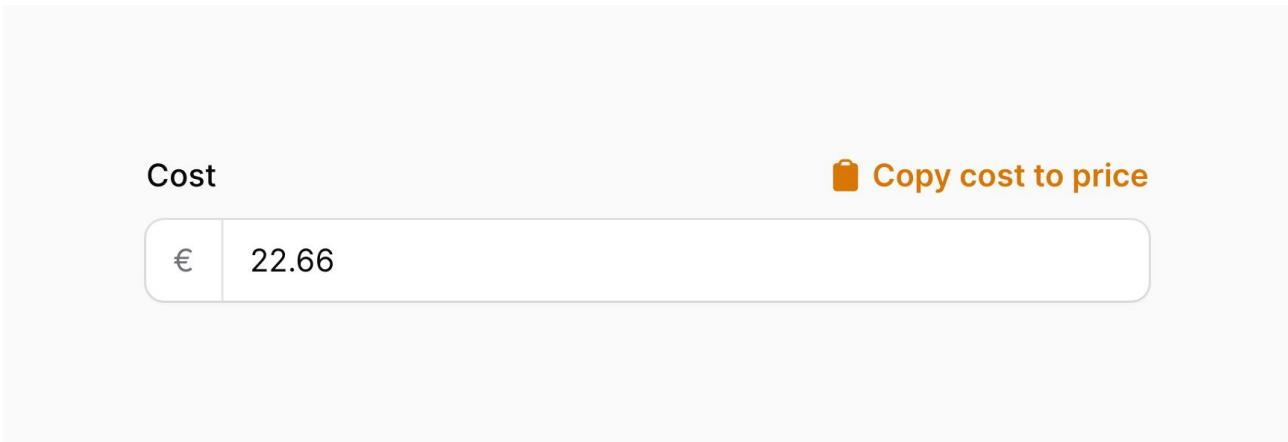
Adding a hint action to a field

All fields support "hint actions", which are rendered aside the field's `hint`. To add a hint action to a field, you may pass it to `hintAction()`:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Set;

TextInput::make('cost')
    ->prefix('€')
    ->hintAction(
        Action::make('copyCostToPrice')
            ->icon('heroicon-m-clipboard')
            ->requiresConfirmation()
            ->action(function (Set $set, $state) {
                $set('price', $state);
            })
    )
)
```

Notice `$set` and `$state` injected into the `action()` function in this example. This is [form component action utility injection](#).



Passing multiple hint actions to a field

You may pass multiple hint actions to a field by passing them in an array to `hintActions()`. Filament will render all the registered actions in order:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\TextInput;

TextInput::make('cost')
    ->prefix('€')
    ->hintActions([
        Action::make('...'),
        Action::make('...'),
        Action::make('...'),
    ])
)
```

Adding an action to a custom form component

If you wish to render an action within a custom form component, `ViewField` object, or `View` component object, you may do so using the `registerActions()` method:

```

use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\ViewField;
use Filament\Forms\Set;

ViewField::make('rating')
    ->view('filament.forms.components.range-slider')
    ->registerActions([
        Action::make('setMaximum')
            ->icon('heroicon-m-star')
            ->action(function (Set $set) {
                $set('rating', 5);
            }),
    ])
)

```

Notice `$set` injected into the `action()` function in this example. This is [form component action utility injection](#).

Now, to render the action in the view of the custom component, you need to call `[$getAction()]`, passing the name of the action you registered:

```

<div x-data="{ state: $wire.$entangle('{{ $getStatePath() }}') }">
    <input x-model="state" type="range" />

    {{ $getAction('setMaximum') }}
</div>

```

Adding "anonymous" actions to a form without attaching them to a component

You may use an `Actions` component to render a set of actions anywhere in the form, avoiding the need to register them to any particular component:

```

use App\Actions\Star;
use App\Actions\ResetStars;
use Filament\Forms\Components\Actions;
use Filament\Forms\Components\Actions\Action;

Actions::make([
    Action::make('star')
        ->icon('heroicon-m-star')
        ->requiresConfirmation()
        ->action(function (Star $star) {
            $star();
        }),
    Action::make('resetStars')
        ->icon('heroicon-m-x-mark')
        ->color('danger')
        ->requiresConfirmation()
        ->action(function (ResetStars $resetStars) {
            $resetStars();
        }),
])
,
```

Making the independent form actions consume the full width of the form

You can stretch the independent form actions to consume the full width of the form using `fullWidth()`:

```
use Filament\Forms\Components\Actions;

Actions::make([
    // ...
])->fullWidth(),
```

Controlling the horizontal alignment of independent form actions

Independent form actions are aligned to the start of the component by default. You may change this by passing `Alignment::Center` or `Alignment::End` to `alignment()`:

```
use Filament\Forms\Components\Actions;
use Filament\Support\Enums\Alignment;

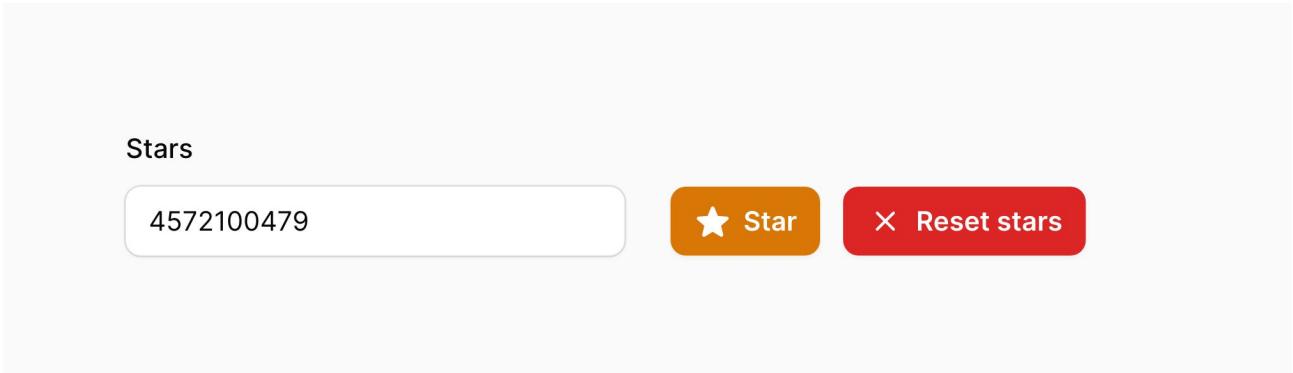
Actions::make([
    // ...
])->alignment(Alignment::Center),
```

Controlling the vertical alignment of independent form actions

Independent form actions are vertically aligned to the start of the component by default. You may change this by passing `Alignment::Center` or `Alignment::End` to `verticalAlignment()`:

```
use Filament\Forms\Components\Actions;
use Filament\Support\Enums\VerticalAlignment;

Actions::make([
    // ...
])->verticalAlignment(VerticalAlignment::End),
```



Form component action utility injection

If an action is attached to a form component, the `action()` function is able to inject utilities directly from that form component. For instance, you can inject `$set` and `$state`:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Set;

Action::make('copyCostToPrice')
    ->icon('heroicon-m-clipboard')
    ->requiresConfirmation()
    ->action(function (Set $set, $state) {
        $set('price', $state);
    })
}
```

Form component actions also have access to all utilities that apply to actions in general.

Advanced

Overview

Filament Form Builder is designed to be flexible and customizable. Many existing form builders allow users to define a form schema, but don't provide a great interface for defining inter-field interactions, or custom logic. Since all Filament forms are built on top of [Livewire](#), the form can adapt dynamically to user input, even after it has been initially rendered. Developers can use [parameter injection](#) to access many utilities in real time and build dynamic forms based on user input. The [lifecycle](#) of fields is open to extension using hook functions to define custom functionality for each field. This allows developers to build complex forms with ease.

The basics of reactivity

[Livewire](#) is a tool that allows Blade-rendered HTML to dynamically re-render without requiring a full page reload. Filament forms are built on top of Livewire, so they are able to re-render dynamically, allowing their layout to adapt after they are initially rendered.

By default, when a user uses a field, the form will not re-render. Since rendering requires a round-trip to the server, this is a performance optimization. However, if you wish to re-render the form after the user has interacted with a field, you can use the `live()` method:

```
use Filament\Forms\Components>Select;

Select::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->live()
```

In this example, when the user changes the value of the `status` field, the form will re-render. This allows you to then make changes to fields in the form based on the new value of the `status` field. Also, you can [hook in to the field's lifecycle](#) to perform custom logic when the field is updated.

Reactive fields on blur

By default, when a field is set to `live()`, the form will re-render every time the field is interacted with. However, this may not be appropriate for some fields like the text input, since making network requests while the user is still typing results in suboptimal performance. You may wish to re-render the form only after the user has finished using the field, when it becomes out of focus. You can do this using the `live(onBlur: true)` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('username')
    ->live(onBlur: true)
```

Debouncing reactive fields

You may wish to find a middle ground between `live()` and `live(onBlur: true)`, using "debouncing". Debouncing will prevent a network request from being sent until a user has finished typing for a certain period of time. You can do this using the `live(debounce: 500)` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('username')
    ->live(debounce: 500) // Wait 500ms before re-rendering the form.
```

In this example, `500` is the number of milliseconds to wait before sending a network request. You can customize this number to whatever you want, or even use a string like `'1s'`.

Form component utility injection

The vast majority of methods used to configure fields and layout components accept functions as parameters instead of hardcoded values:

```
use App\Models\User;
use Filament\Forms\Components\DatePicker;
use Filament\Forms\Components>Select;
use Filament\Forms\Components\TextInput;

DatePicker::make('date_of_birth')
    ->displayFormat(function (): string {
        if ($auth()->user()->country_id === 'us') {
            return 'm/d/Y';
        }

        return 'd/m/Y';
    })

Select::make('user_id')
    ->options(function (): array {
        return User::all()->pluck('name', 'id')->all();
    })

TextInput::make('middle_name')
    ->required(fn (): bool => $auth()->user()->hasMiddleName())
```

This alone unlocks many customization possibilities.

The package is also able to inject many utilities to use inside these functions, as parameters. All customization methods that accept functions as arguments can inject utilities.

These injected utilities require specific parameter names to be used. Otherwise, Filament doesn't know what to inject.

Injecting the current state of a field

If you wish to access the current state (value) of the field, define a `$state` parameter:

```
function ($state) {
    // ...
}
```

Injecting the current form component instance

If you wish to access the current component instance, define a `$component` parameter:

```
use Filament\Forms\Components\Component;

function (Component $component) {
    // ...
}
```

Injecting the current Livewire component instance

If you wish to access the current Livewire component instance, define a `$livewire` parameter:

```
use Livewire\Component as Livewire;

function (Livewire $livewire) {
    // ...
}
```

Injecting the current form record

If your form is associated with an Eloquent model instance, define a `$record` parameter:

```
use Illuminate\Database\Eloquent\Model;

function (?Model $record) {
    // ...
}
```

Injecting the state of another field

You may also retrieve the state (value) of another field from within a callback, using a `$get` parameter:

```
use Filament\Forms\Get;

function (Get $get) {
    $email = $get('email'); // Store the value of the `email` field in the `$email` variable.
    //...
}
```

Injecting a function to set the state of another field

In a similar way to `$get`, you may also set the value of another field from within a callback, using a `$set` parameter:

```
use Filament\Forms\Set;

function (Set $set) {
    $set('title', 'Blog Post'); // Set the `title` field to `Blog Post`.
    //...
}
```

When this function is run, the state of the `title` field will be updated, and the form will re-render with the new title. This is useful inside the `afterStateUpdated` method.

Injecting the current form operation

If you're writing a form for a panel resource or relation manager, and you wish to check if a form is `create`, `edit` or `view`, use the `$operation` parameter:

```
function (string $operation) {
    // ...
}
```

Outside the panel, you can set a form's operation by using the `operation()` method on the form definition.

Injecting multiple utilities

The parameters are injected dynamically using reflection, so you are able to combine multiple parameters in any order:

```
use Filament\Forms\Get;
use Filament\Forms\Set;
use Livewire\Component as Livewire;

function (Livewire $livewire, Get $get, Set $set) {
    // ...
}
```

Injecting dependencies from Laravel's container

You may inject anything from Laravel's container like normal, alongside utilities:

```
use Filament\Forms\Set;
use Illuminate\Http\Request;

function (Request $request, Set $set) {
    // ...
}
```

Field lifecycle

Each field in a form has a lifecycle, which is the process it goes through when the form is loaded, when it is interacted with by the user, and when it is submitted. You may customize what happens at each stage of this lifecycle using a function that gets run at that stage.

Field hydration

Hydration is the process that fills fields with data. It runs when you call the form's `fill()` method. You may customize what happens after a field is hydrated using the `afterStateHydrated()` method.

In this example, the `name` field will always be hydrated with the correctly capitalized name:

```
use Closure;
use Filament\Forms\Components\TextInput;

TextInput::make('name')
    ->required()
    ->afterStateHydrated(function (TextInput $component, string $state) {
        $component->state(ucwords($state));
    })
}
```

As a shortcut for formatting the field's state like this when it is hydrated, you can use the `formatStateUsing()` method:

```
use Closure;
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->formatStateUsing(fn (string $state): string => ucwords($state))
```

Field updates

You may use the `afterStateUpdated()` method to customize what happens after a field is updated by the user. Only changes from the user on the frontend will trigger this function, not manual changes to the state from `$set()` or another PHP function.

Inside this function, you can also inject the `$old` value of the field before it was updated, using the `$old` parameter:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->afterStateUpdated(function (?string $state, ?string $old) {
    // ...
})
```

For an example of how to use this method, learn how to [automatically generate a slug from a title](#).

Field dehydration

Dehydration is the process that gets data from the fields in your forms, and transforms it. It runs when you call the form's `getState()` method.

You may customize how the state is transformed when it is dehydrated using the `dehydrateStateUsing()` function. In this example, the `name` field will always be dehydrated with the correctly capitalized name:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->required()
->dehydrateStateUsing(fn (string $state): string => ucwords($state))
```

Preventing a field from being dehydrated

You may also prevent a field from being dehydrated altogether using `dehydrated(false)`. In this example, the field will not be present in the array returned from `getState()`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password_confirmation')
->password()
->dehydrated(false)
```

If your form auto-saves data to the database, like in a [resource](#) or [table](#) action, this is useful to prevent a field from being saved to the database if it is purely used for presentational purposes.

Reactive forms cookbook

This section contains a collection of recipes for common tasks you may need to perform when building an advanced form.

Conditionally hiding a field

To conditionally hide or show a field, you can pass a function to the `hidden()` method, and return `true` or `false` depending on whether you want the field to be hidden or not. The function can inject utilities as parameters, so you can do things like check the value of another field:

```
use Filament\Forms\Get;
use Filament\Forms\Components\Checkbox;
use Filament\Forms\Components\TextInput;

Checkbox::make('is_company')
->live()

TextInput::make('company_name')
->hidden(fn (Get $get): bool => ! $get('is_company'))
```

In this example, the `is_company` checkbox is `live()`. This allows the form to rerender when the value of the `is_company` field changes. You can access the value of that field from within the `hidden()` function using the `$get()` utility. The value of the field is inverted using `!` so that the `company_name` field is hidden when the `is_company` field is `false`.

Alternatively, you can use the `visible()` method to show a field conditionally. It does the exact inverse of `hidden()`, and could be used if you prefer the clarity of the code when written this way:

```
use Filament\Forms\Get;
use Filament\Forms\Components\Checkbox;
use Filament\Forms\Components\TextInput;

Checkbox::make('is_company')
->live()

TextInput::make('company_name')
->visible(fn (Get $get): bool => $get('is_company'))
```

Conditionally making a field required

To conditionally make a field required, you can pass a function to the `required()` method, and return `true` or `false` depending on whether you want the field to be required or not. The function can inject utilities as parameters, so you can do things like check the value of another field:

```
use Filament\Forms\Get;
use Filament\Forms\Components\TextInput;

TextInput::make('company_name')
->live(onBlur: true)

TextInput::make('vat_number')
->required(fn (Get $get): bool => filled($get('company_name')))
```

In this example, the `company_name` field is `live(onBlur: true)`. This allows the form to rerender after the value of the `company_name` field changes and the user clicks away. You can access the value of that field from within the `required()` function using the `$get()` utility. The value of the field is checked using `filled()` so that the

`vat_number` field is required when the `company_name` field is not `null` or an empty string. The result is that the `vat_number` field is only required when the `company_name` field is filled in.

Using a function is able to make any other [validation rule](#) dynamic in a similar way.

Generating a slug from a title

To generate a slug from a title while the user is typing, you can use the `afterStateUpdated()` method on the title field to `$set()` the value of the slug field:

```
use Filament\Forms\Components\TextInput;
use Filament\Forms\Set;
use Illuminate\Support\Str;

TextInput::make('title')
    ->live()
    ->afterStateUpdated(fn (Set $set, ?string $state) => $set('slug', Str::slug($state)))

TextInput::make('slug')
```

In this example, the `title` field is `live()`. This allows the form to rerender when the value of the `title` field changes. The `afterStateUpdated()` method is used to run a function after the state of the `title` field is updated. The function injects the `$set()` utility and the new state of the `title` field. The `Str::slug()` utility method is part of Laravel and is used to generate a slug from a string. The `slug` field is then updated using the `$set()` function.

One thing to note is that the user may customize the slug manually, and we don't want to overwrite their changes if the title changes. To prevent this, we can use the old version of the title to work out if the user has modified it themselves. To access the old version of the title, you can inject `$old`, and to get the current value of the slug before it gets changed, we can use the `$get()` utility:

```
use Filament\Forms\Components\TextInput;
use Filament\Forms\Get;
use Filament\Forms\Set;
use Illuminate\Support\Str;

TextInput::make('title')
    ->live()
    ->afterStateUpdated(function (Get $get, Set $set, ?string $old, ?string $state) {
        if (($get('slug') ?? '') !== Str::slug($old)) {
            return;
        }

        $set('slug', Str::slug($state));
    })

TextInput::make('slug')
```

Dependant select options

To dynamically update the options of a [select field](#) based on the value of another field, you can pass a function to the `options()` method of the select field. The function can [inject utilities](#) as parameters, so you can do things like check the value of another field using the `$get()` utility:

```

use Filament\Forms\Get;
use Filament\Forms\Components>Select;

Select::make('category')
    ->options([
        'web' => 'Web development',
        'mobile' => 'Mobile development',
        'design' => 'Design',
    ])
    ->live()

Select::make('sub_category')
    ->options(fn (Get $get): array => match ($get('category')) {
        'web' => [
            'frontend_web' => 'Frontend development',
            'backend_web' => 'Backend development',
        ],
        'mobile' => [
            'ios_mobile' => 'iOS development',
            'android_mobile' => 'Android development',
        ],
        'design' => [
            'app_design' => 'Panel design',
            'marketing_website_design' => 'Marketing website design',
        ],
        default => [],
    })
}

```

In this example, the `category` field is `live()`. This allows the form to rerender when the value of the `category` field changes. You can access the value of that field from within the `options()` function using the `$get()` utility. The value of the field is used to determine which options should be available in the `sub_category` field. The `match ()` statement in PHP is used to return an array of options based on the value of the `category` field. The result is that the `sub_category` field will only show options relevant to the selected `category` field.

You could adapt this example to use options loaded from an Eloquent model or other data source, by querying within the function:

```

use Filament\Forms\Get;
use Filament\Forms\Components>Select;
use Illuminate\Support\Collection;

Select::make('category')
    ->options(Category::query()->pluck('name', 'id'))
    ->live()

Select::make('sub_category')
    ->options(fn (Get $get): Collection => SubCategory::query()
        ->where('category', $get('category'))
        ->pluck('name', 'id'))

```

Dynamic fields based on a select option

You may wish to render a different set of fields based on the value of a field, like a select. To do this, you can pass a function to the `schema()` method of any layout component, which checks the value of the field and returns a different

schema based on that value. Also, you will need a way to initialise the new fields in the dynamic schema when they are first loaded.

```
use Filament\Forms\Components\FileUpload;
use Filament\Forms\Components\Grid;
use Filament\Forms\Components\Select;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Get;

Select::make('type')
    ->options([
        'employee' => 'Employee',
        'freelancer' => 'Freelancer',
    ])
    ->live()
    ->afterStateUpdated(fn (Select $component) => $component
        ->getContainer()
        ->getComponent('dynamicTypeFields')
        ->getChildComponentContainer()
        ->fill()))

Grid::make(2)
    ->schema(fn (Get $get): array => match ($get('type')) {
        'employee' => [
            TextInput::make('employee_number')
                ->required(),
            FileUpload::make('badge')
                ->image()
                ->required(),
        ],
        'freelancer' => [
            TextInput::make('hourly_rate')
                ->numeric()
                ->required()
                ->prefix('€'),
            FileUpload::make('contract')
                ->required(),
        ],
        default => [],
    })
    ->key('dynamicTypeFields')
```

In this example, the `type` field is `live()`. This allows the form to rerender when the value of the `type` field changes. The `afterStateUpdated()` method is used to run a function after the state of the `type` field is updated. In this case, we inject the current select field instance, which we can then use to get the form "container" instance that holds both the select and the grid components. With this container, we can target the grid component using a unique key (`'dynamicTypeFields'`) that we have assigned to it. With that grid component instance, we can call `fill()`, just as we do on a normal form to initialise it. The `schema()` method of the grid component is then used to return a different schema based on the value of the `type` field. This is done by using the `$get()` utility, and returning a different schema array dynamically.

Auto-hashing password field

You have a password field:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
->password()
```

And you can use a dehydration function to hash the password when the form is submitted:

```
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Hash;

TextInput::make('password')
->password()
->dehydrateStateUsing(fn (string $state): string => Hash::make($state))
```

But if your form is used to change an existing password, you don't want to overwrite the existing password if the field is empty. You can prevent the field from being dehydrated if the field is null or an empty string (using the `filled()` helper):

```
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Hash;

TextInput::make('password')
->password()
->dehydrateStateUsing(fn (string $state): string => Hash::make($state))
->dehydrated(fn (?string $state): bool => filled($state))
```

However, you want to require the password to be filled when the user is being created, by injecting the `$operation` utility, and then conditionally making the field required:

```
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Hash;

TextInput::make('password')
->password()
->dehydrateStateUsing(fn (string $state): string => Hash::make($state))
->dehydrated(fn (?string $state): bool => filled($state))
->required(fn (string $operation): bool => $operation === 'create')
```

Saving data to relationships

If you're building a form inside your Livewire component, make sure you have set up the form's model. Otherwise, Filament doesn't know which model to use to retrieve the relationship from.

As well as being able to give structure to fields, layout components are also able to "teleport" their nested fields into a relationship. Filament will handle loading data from a `HasOne`, `BelongsTo` or `MorphOne` Eloquent relationship, and then it will save the data back to the same relationship. To set this behavior up, you can use the `relationship()` method on any layout component:

```
use Filament\Forms\Components\Fieldset;
use Filament\Forms\Components\FileUpload;
use Filament\Forms\Components\Textarea;
use Filament\Forms\Components\TextInput;

Fieldset::make('Metadata')
    ->relationship('metadata')
    ->schema([
        TextInput::make('title'),
        Textarea::make('description'),
        FileUpload::make('image'),
    ])
)
```

In this example, the `title`, `description` and `image` are automatically loaded from the `metadata` relationship, and saved again when the form is submitted. If the `metadata` record does not exist, it is automatically created.

This functionality is not just limited to fieldsets - you can use it with any layout component. For example, you could use a `Group` component which has no styling associated with it:

```
use Filament\Forms\Components\Group;
use Filament\Forms\Components\TextInput;

Group::make()
    ->relationship('customer')
    ->schema([
        TextInput::make('name')
            ->label('Customer')
            ->required(),
        TextInput::make('email')
            ->label('Email address')
            ->email()
            ->required(),
    ])
)
```

Saving data to a `BelongsTo` relationship

Please note that if you are saving the data to a `BelongsTo` relationship, then the foreign key column in your database must be `nullable()`. This is because Filament saves the form first, before saving the relationship. Since the form is saved first, the foreign ID does not exist yet, so it must be nullable. Immediately after the form is saved, Filament saves the relationship, which will then fill in the foreign ID and save it again.

It is worth noting that if you have an observer on your form model, then you may need to adapt it to ensure that it does not depend on the relationship existing when it is created. For example, if you have an observer that sends an email to a related record when a form is created, you may need to switch to using a different hook that runs after the relationship is attached, like `updated()`.

Conditionally saving data to a relationship

Sometimes, saving the related record may be optional. If the user fills out the customer fields, then the customer will be created / updated. Otherwise, the customer will not be created, or will be deleted if it already exists. To do this, you can pass a `condition` function as an argument to `relationship()`, which can use the `$state` of the related form to determine whether the relationship should be saved or not:

```
use Filament\Forms\Components\Group;
use Filament\Forms\Components\TextInput;

Group::make()
    ->relationship(
        'customer',
        condition: fn (?array $state): bool => filled($state['name']),
    )
    ->schema([
        TextInput::make('name')
            ->label('Customer'),
        TextInput::make('email')
            ->label('Email address')
            ->email()
            ->requiredWith('name'),
    ])
])
```

In this example, the customer's name is not `required()`, and the email address is only required when the `name` is filled. The `condition` function is used to check whether the `name` field is filled, and if it is, then the customer will be created / updated. Otherwise, the customer will not be created, or will be deleted if it already exists.

Inserting Livewire components into a form

You may insert a Livewire component directly into a form:

```
use Filament\Forms\Components\Livewire;
use App\LiveWire\Foo;

Livewire::make(Foo::class)
```

If you are rendering multiple of the same Livewire component, please make sure to pass a unique `key()` to each:

```
use Filament\Forms\Components\Livewire;
use App\LiveWire\Foo;

Livewire::make(Foo::class)
    ->key('foo-first')

Livewire::make(Foo::class)
    ->key('foo-second')

Livewire::make(Foo::class)
    ->key('foo-third')
```

Passing parameters to a Livewire component

You can pass an array of parameters to a Livewire component:

```
use Filament\Forms\Components\Livewire;
use App\LiveWire\Foo;

Livewire::make(Foo::class, ['bar' => 'baz'])
```

Now, those parameters will be passed to the Livewire component's `mount()` method:

```
class Foo extends Component
{
    public function mount(string $bar): void
    {
        // ...
    }
}
```

Alternatively, they will be available as public properties on the Livewire component:

```
class Foo extends Component
{
    public string $bar;
}
```

Accessing the current record in the Livewire component

You can access the current record in the Livewire component using the `$record` parameter in the `mount()` method, or the `$record` property:

```
use Illuminate\Database\Eloquent\Model;

class Foo extends Component
{
    public function mount(?Model $record = null): void
    {
        // ...
    }

    // or

    public ?Model $record = null;
}
```

Please be aware that when the record has not yet been created, it will be `null`. If you'd like to hide the Livewire component when the record is `null`, you can use the `hidden()` method:

```
use Filament\Forms\Components\Livewire;
use Illuminate\Database\Eloquent\Model;

Livewire::make(Foo::class)
    ->hidden(fn (?Model $record): bool => $record === null)
```

Lazy loading a Livewire component

You may allow the component to lazily load using the `lazy()` method:

```
use Filament\Forms\Components\Livewire;
use App\LiveWire\Foo;

Livewire::make(Foo::class)->lazy()
```

Adding A Form To A Livewire Component

Setting up the Livewire component

First, generate a new Livewire component:

```
php artisan make:livewire CreatePost
```

Then, render your Livewire component on the page:

```
@livewire('create-post')
```

Alternatively, you can use a full-page Livewire component:

```
use App\LiveWire\CreatePost;
use Illuminate\Support\Facades\Route;

Route::get('posts/create', CreatePost::class);
```

Adding the form

There are 5 main tasks when adding a form to a Livewire component class. Each one is essential:

1. Implement the `HasForms` interface and use the `InteractsWithForms` trait.
2. Define a public Livewire property to store your form's data. In our example, we'll call this `$data`, but you can call it whatever you want.
3. Add a `form()` method, which is where you configure the form. Add the form's schema, and tell Filament to store the form data in the `$data` property (using `statePath('data')`).
4. Initialize the form with `$this->form->fill()` in `mount()`. This is imperative for every form that you build, even if it doesn't have any initial data.
5. Define a method to handle the form submission. In our example, we'll call this `create()`, but you can call it whatever you want. Inside that method, you can validate and get the form's data using `$this->form->getState()`. It's important that you use this method instead of accessing the `$this->data` property directly, because the form's data needs to be validated and transformed into a useful format before being returned.

```

<?php

namespace App\Livewire;

use App\Models\Post;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Concerns\InteractsWithForms;
use Filament\Forms\Contracts\HasForms;
use Filament\Forms\Form;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class CreatePost extends Component implements HasForms
{
    use InteractsWithForms;

    public ?array $data = [];

    public function mount(): void
    {
        $this->form->fill();
    }

    public function form(Form $form): Form
    {
        return $form
            ->schema([
                TextInput::make('title')
                    ->required(),
                MarkdownEditor::make('content'),
                // ...
            ])
            ->statePath('data');
    }

    public function create(): void
    {
        dd($this->form->getState());
    }

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

```

Finally, in your Livewire component's view, render the form:

```
<div>
    <form wire:submit="create">
        {{ $this->form }}

        <button type="submit">
            Submit
        </button>
    </form>

    <x-filament-actions::modals />
</div>
```

`<x-filament-actions::modals />` is used to render form component action modals. The code can be put anywhere outside the `<form>` element, as long as it's within the Livewire component.

Visit your Livewire component in the browser, and you should see the form components from `schema()`:

Submit the form with data, and you'll see the form's data dumped to the screen. You can save the data to a model instead of dumping it:

```
use App\Models\Post;

public function create(): void
{
    Post::create($this->form->getState());
}
```

Initializing the form with data

To fill the form with data, just pass that data to the `$this->form->fill()` method. For example, if you're editing an existing post, you might do something like this:

```
use App\Models\Post;

public function mount(Post $post): void
{
    $this->form->fill($post->toArray());
}
```

It's important that you use the `$this->form->fill()` method instead of assigning the data directly to the `$this->data` property. This is because the post's data needs to be internally transformed into a useful format before being stored.

Setting a form model

Giving the `$form` access to a model is useful for a few reasons:

- It allows fields within that form to load information from that model. For example, select fields can load their options from the database automatically.
- The form can load and save the model's relationship data automatically. For example, you have an Edit Post form, with a Repeater which manages comments associated with that post. Filament will automatically load the comments for that post when you call `$this->form->fill([...])`, and save them back to the relationship when you call `$this->form->getState()`.

- Validation rules like `exists()` and `unique()` can automatically retrieve the database table name from the model.

It is advised to always pass the model to the form when there is one. As explained, it unlocks many new powers of the Filament Form Builder.

To pass the model to the form, use the `$form->model()` method:

```
use App\Models\Post;
use Filament\Forms\Form;

public Post $post;

public function form(Form $form): Form
{
    return $form
        ->schema([
            // ...
        ])
        ->statePath('data')
        ->model($this->post);
}
```

Passing the form model after the form has been submitted

In some cases, the form's model is not available until the form has been submitted. For example, in a Create Post form, the post does not exist until the form has been submitted. Therefore, you can't pass it in to `$form->model()`. However, you can pass a model class instead:

```
use App\Models\Post;
use Filament\Forms\Form;

public function form(Form $form): Form
{
    return $form
        ->schema([
            // ...
        ])
        ->statePath('data')
        ->model(Post::class);
}
```

On its own, this isn't as powerful as passing a model instance. For example, relationships won't be saved to the post after it is created. To do that, you'll need to pass the post to the form after it has been created, and call `saveRelationships()` to save the relationships to it:

```
use App\Models\Post;

public function create(): void
{
    $post = Post::create($this->form->getState());

    // Save the relationships from the form to the post after it is created.
    $this->form->model($post)->saveRelationships();
}
```

Saving form data to individual properties

In all of our previous examples, we've been saving the form's data to the public `$data` property on the Livewire component. However, you can save the data to individual properties instead. For example, if you have a form with a `title` field, you can save the form's data to the `$title` property instead. To do this, don't pass a `statePath()` to the form at all. Ensure that all of your fields have their own **public** properties on the class.

```
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Form;

public ?string $title = null;

public ?string $content = null;

public function form(Form $form): Form
{
    return $form
        ->schema([
            TextInput::make('title')
                ->required(),
            MarkdownEditor::make('content'),
            // ...
        ]);
}
```

Using multiple forms

By default, the `InteractsWithForms` trait only handles one form per Livewire component - `form()`. To add more forms to the Livewire component, you can define them in the `getForms()` method, and return an array containing the name of each form:

```
protected function getForms(): array
{
    return [
        'editPostForm',
        'createCommentForm',
    ];
}
```

Each of these forms can now be defined within the Livewire component, using a method with the same name:

```

use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Form;

public function editPostForm(Form $form): Form
{
    return $form
        ->schema([
            TextInput::make('title')
                ->required(),
            MarkdownEditor::make('content'),
            // ...
        ])
        ->statePath('postData')
        ->model($this->post);
}

public function createCommentForm(Form $form): Form
{
    return $form
        ->schema([
            TextInput::make('name')
                ->required(),
            TextInput::make('email')
                ->email()
                ->required(),
            MarkdownEditor::make('content')
                ->required(),
            // ...
        ])
        ->statePath('commentData')
        ->model(Comment::class);
}

```

Now, each form is addressable by its name instead of `form`. For example, to fill the post form, you can use `$this->editPostForm->fill([...])`, or to get the data from the comment form you can use `$this->createCommentForm->getState()`.

You'll notice that each form has its own unique `statePath()`. Each form will write its state to a different array on your Livewire component, so it's important to define these:

```

public ?array $postData = [];
public ?array $commentData = [];

```

Resetting a form's data

You can reset a form back to its default data at any time by calling `$this->form->fill()`. For example, you may wish to clear the contents of a form every time it's submitted:

```
use App\Models\Comment;

public function createComment(): void
{
    Comment::create($this->form->getState());

    // Reinitialize the form to clear its data.
    $this->form->fill();
}
```

Generating form Livewire components with the CLI

It's advised that you learn how to set up a Livewire component with the Form Builder manually, but once you are confident, you can use the CLI to generate a form for you.

```
php artisan make:livewire-form RegistrationForm
```

This will generate a new `app/Livewire/RegistrationForm.php` component, which you can customize.

Generating a form for an Eloquent model

Filament is also able to generate forms for a specific Eloquent model. These are more powerful, as they will automatically save the data in the form for you, and [ensure the form fields are properly configured](#) to access that model.

When generating a form with the `make:livewire-form` command, it will ask for the name of the model:

```
php artisan make:livewire-form Products/CreateProduct
```

Generating an edit form for an Eloquent record

By default, passing a model to the `make:livewire-form` command will result in a form that creates a new record in your database. If you pass the `--edit` flag to the command, it will generate an edit form for a specific record. This will automatically fill the form with the data from the record, and save the data back to the model when the form is submitted.

```
php artisan make:livewire-form Products/EditProduct --edit
```

Automatically generating form schemas

Filament is also able to guess which form fields you want in the schema, based on the model's database columns. You can use the `--generate` flag when generating your form:

```
php artisan make:livewire-form Products/CreateProduct --generate
```

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Since the Form Builder works on Livewire components, you can use the [Livewire testing helpers](#). However, we have custom testing helpers that you can use with forms:

Filling a form

To fill a form with data, pass the data to `fillForm()`:

```
use function Pest\LiveWire\livewire;

livewire(CreatePost::class)
    ->fillForm([
        'title' => fake() ->sentence(),
        // ...
    ]);
```

If you have multiple forms on a Livewire component, you can specify which form you want to fill using

```
fillForm([...], 'createPostForm');
```

To check that a form has data, use `assertFormSet()`:

```
use Illuminate\Support\Str;
use function Pest\LiveWire\livewire;

it('can automatically generate a slug from the title', function () {
    $title = fake() ->sentence();

    livewire(CreatePost::class)
        ->fillForm([
            'title' => $title,
        ])
        ->assertFormSet([
            'slug' => Str::slug($title),
        ]);
});
```

If you have multiple forms on a Livewire component, you can specify which form you want to check using

```
assertFormSet([...], 'createPostForm');
```

You may also find it useful to pass a function to the `assertFormSet()` method, which allows you to access the form `$state` and perform additional assertions:

```

use Illuminate\Support\Str;
use function Pest\Livewire\livewire;

it('can automatically generate a slug from the title without any spaces', function () {
    $title = fake()->sentence();

    livewire(CreatePost::class)
        ->fillForm([
            'title' => $title,
        ])
        ->assertFormSet(function (array $state): array {
            expect($state['slug'])
                ->not->toContain(' ');
        });

        return [
            'slug' => Str::slug($title),
        ];
});
});

```

You can return an array from the function if you want Filament to continue to assert the form state after the function has been run.

Validation

Use `assertHasFormErrors()` to ensure that data is properly validated in a form:

```

use function Pest\Livewire\livewire;

it('can validate input', function () {
    livewire(CreatePost::class)
        ->fillForm([
            'title' => null,
        ])
        ->call('create')
        ->assertHasFormErrors(['title' => 'required']);
});

```

And `assertHasNoFormErrors()` to ensure there are no validation errors:

```

use function Pest\Livewire\livewire;

livewire(CreatePost::class)
    ->fillForm([
        'title' => fake()->sentence(),
        // ...
    ])
    ->call('create')
    ->assertHasNoFormErrors();

```

If you have multiple forms on a Livewire component, you can pass the name of a specific form as the second parameter like `assertHasFormErrors(['title' => 'required'], 'createPostForm')` or `assertHasNoFormErrors([], 'createPostForm')`.

Form existence

To check that a Livewire component has a form, use `assertFormExists()`:

```
use function Pest\Livewire\livewire;

it('has a form', function () {
    livewire(CreatePost::class)
        ->assertFormExists();
});
```

If you have multiple forms on a Livewire component, you can pass the name of a specific form like `assertFormExists('createPostForm')`.

Fields

To ensure that a form has a given field, pass the field name to `assertFormFieldExists()`:

```
use function Pest\Livewire\livewire;

it('has a title field', function () {
    livewire(CreatePost::class)
        ->assertFormFieldExists('title');
});
```

You may pass a function as an additional argument in order to assert that a field passes a given "truth test". This is useful for asserting that a field has a specific configuration:

```
use function Pest\Livewire\livewire;

it('has a title field', function () {
    livewire(CreatePost::class)
        ->assertFormFieldExists('title', function (TextInput $field): bool {
            return $field->isDisabled();
        });
});
```

To assert that a form does not have a given field, pass the field name to `assertFormFieldDoesNotExist()`:

```
use function Pest\Livewire\livewire;

it('does not have a conditional field', function () {
    livewire(CreatePost::class)
        ->assertFormFieldDoesNotExist('no-such-field');
});
```

If you have multiple forms on a Livewire component, you can specify which form you want to check for the existence of the field like `assertFormFieldExists('title', 'createPostForm')`.

Hidden fields

To ensure that a field is visible, pass the name to `assertFormFieldIsVisible()`:

```
use function Pest\LiveWire\livewire;

test('title is visible', function () {
    livewire(CreatePost::class)
        ->assertFormFieldIsVisible('title');
});
```

Or to ensure that a field is hidden you can pass the name to `assertFormFieldIsHidden()`:

```
use function Pest\LiveWire\livewire;

test('title is hidden', function () {
    livewire(CreatePost::class)
        ->assertFormFieldIsHidden('title');
});
```

For both `assertFormFieldIsHidden()` and `assertFormFieldIsVisible()` you can pass the name of a specific form the field belongs to as the second argument like `assertFormFieldIsHidden('title', 'createPostForm')`.

Disabled fields

To ensure that a field is enabled, pass the name to `assertFormFieldIsEnabled()`:

```
use function Pest\LiveWire\livewire;

test('title is enabled', function () {
    livewire(CreatePost::class)
        ->assertFormFieldIsEnabled('title');
});
```

Or to ensure that a field is disabled you can pass the name to `assertFormFieldIsDisabled()`:

```
use function Pest\LiveWire\livewire;

test('title is disabled', function () {
    livewire(CreatePost::class)
        ->assertFormFieldIsDisabled('title');
});
```

For both `assertFormFieldIsEnabled()` and `assertFormFieldIsDisabled()` you can pass the name of a specific form the field belongs to as the second argument like `assertFormFieldIsEnabled('title', 'createPostForm')`.

Layout components

If you need to check if a particular layout component exists rather than a field, you may use `assertFormComponentExists()`. As layout components do not have names, this method uses the `key()` provided by the developer:

```
use Filament\Forms\Components\Section;

Section::make('Comments')
    ->key('comments-section')
    ->schema([
        //
    ])
});
```

```
use function Pest\Livewire\livewire;

test('comments section exists' function () {
    livewire(EditPost::class)
        ->assertFormComponentExists('comments-section');
});
```

To assert that a form does not have a given component, pass the component key to

```
assertFormComponentDoesNotExist();
```

```
use function Pest\Livewire\livewire;

it('does not have a conditional component', function () {
    livewire(CreatePost::class)
        ->assertFormComponentDoesNotExist('no-such-section');
});
```

To check if the component exists and passes a given truth test, you can pass a function to the second argument of

```
assertFormComponentExists(), returning true or false if the component passes the test or not:
```

```
use Filament\Forms\Components\Component;

use function Pest\Livewire\livewire;

test('comments section has heading' function () {
    livewire(EditPost::class)
        ->assertFormComponentExists(
            'comments-section',
            function (Component $component): bool {
                return $component->getHeading() === 'Comments';
            },
        );
});
```

If you want more informative test results, you can embed an assertion within your truth test callback:

```

use Filament\Forms\Components\Component;
use Illuminate\Testing\Assert;

use function Pest\LiveWire\livewire;

test('comments section is enabled' function () {
    livewire(EditPost::class)
        ->assertFormComponentExists(
            'comments-section',
            function (Component $component): bool {
                Assert::assertTrue(
                    $component->isEnabled(),
                    'Failed asserting that comments-section is enabled.',
                );
            }
        )
        return true;
);
});

```

Actions

You can call an action by passing its form component name, and then the name of the action to

`callFormComponentAction()`:

```

use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->callFormComponentAction('customer_id', 'send');

    expect($invoice->refresh())
        ->isSent()->toBeTrue();
});

```

To pass an array of data into an action, use the `data` parameter:

```
use function Pest\Livewire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callFormComponentAction('customer_id', 'send', data: [
        'email' => $email = fake()->email(),
    ])
    ->assertHasNoFormComponentActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($email);
});

});
```

If you ever need to only set an action's data without immediately calling it, you can use

`setFormComponentActionData()`:

```
use function Pest\Livewire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountFormComponentAction('customer_id', 'send')
    ->setFormComponentActionData('customer_id', 'send', data: [
        'email' => $email = fake()->email(),
    ])
});

});
```

Execution

To check if an action has been halted, you can use `assertFormComponentActionHalted()`:

```
use function Pest\Livewire\livewire;

it('stops sending if invoice has no email address', function () {
    $invoice = Invoice::factory(['email' => null])->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callFormComponentAction('customer_id', 'send')
    ->assertFormComponentActionHalted('customer_id', 'send');
});

});
```

Errors

`assertHasNoFormComponentActionErrors()` is used to assert that no validation errors occurred when submitting the action form.

To check if a validation error has occurred with the data, use `assertHasFormComponentActionErrors()`, similar to `assertHasErrors()` in Livewire:

```
use function Pest\LiveWire\livewire;

it('can validate invoice recipient email', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callFormComponentAction('customer_id', 'send', data: [
        'email' => Str::random(),
    ])
    ->assertHasFormComponentActionErrors(['email' => ['email']]);
});
```

To check if an action is pre-filled with data, you can use the `assertFormComponentActionDataSet()` method:

```
use function Pest\LiveWire\livewire;

it('can send invoices to the primary contact by default', function () {
    $invoice = Invoice::factory()->create();
    $recipientEmail = $invoice->company->primaryContact->email;

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountFormComponentAction('customer_id', 'send')
    ->assertFormComponentActionDataSet([
        'email' => $recipientEmail,
    ])
    ->callMountedFormComponentAction()
    ->assertHasNoFormComponentActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($recipientEmail);
});
```

Action state

To ensure that an action exists or doesn't in a form, you can use the `assertFormComponentActionExists()` or `assertFormComponentActionDoesNotExist()` method:

```
use function Pest\Livewire\livewire;

it('can send but not unsend invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionExists('customer_id', 'send')
    ->assertFormComponentActionDoesNotExist('customer_id', 'unsend');
});
```

To ensure an action is hidden or visible for a user, you can use the `assertFormComponentActionHidden()` or `assertFormComponentActionVisible()` methods:

```
use function Pest\Livewire\livewire;

it('can only print customers', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionHidden('customer_id', 'send')
    ->assertFormComponentActionVisible('customer_id', 'print');
});
```

To ensure an action is enabled or disabled for a user, you can use the `assertFormComponentActionEnabled()` or `assertFormComponentActionDisabled()` methods:

```
use function Pest\Livewire\livewire;

it('can only print a customer for a sent invoice', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionDisabled('customer_id', 'send')
    ->assertFormComponentActionEnabled('customer_id', 'print');
});
```

To check if an action is hidden to a user, you can use the `assertFormComponentActionHidden()` method:

```
use function Pest\Livewire\livewire;

it('can not send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionHidden('customer_id', 'send');
});
```

Button appearance

To ensure an action has the correct label, you can use `assertFormComponentActionHasLabel()` and `assertFormComponentActionDoesNotHaveLabel()`:

```
use function Pest\Livewire\livewire;

it('send action has correct label', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionHasLabel('customer_id', 'send', 'Email Invoice')
    ->assertFormComponentActionDoesNotHaveLabel('customer_id', 'send', 'Send');
});
```

To ensure an action's button is showing the correct icon, you can use `assertFormComponentActionHasIcon()` or `assertFormComponentActionDoesNotHaveIcon()`:

```
use function Pest\Livewire\livewire;

it('when enabled the send button has correct icon', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionEnabled('customer_id', 'send')
    ->assertFormComponentActionHasIcon('customer_id', 'send', 'envelope-open')
    ->assertFormComponentActionDoesNotHaveIcon('customer_id', 'send', 'envelope');
});
```

To ensure that an action's button is displaying the right color, you can use `assertFormComponentActionHasColor()` or `assertFormComponentActionDoesNotHaveColor()`:

```
use function Pest\Livewire\livewire;

it('actions display proper colors', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionHasColor('customer_id', 'delete', 'danger')
    ->assertFormComponentActionDoesNotHaveColor('customer_id', 'print', 'danger');
});
```

URL

To ensure an action has the correct URL, you can use `assertFormComponentActionHasUrl()`, `assertFormComponentActionDoesNotHaveUrl()`, `assertFormComponentActionShouldOpenUrlInNewTab()`, and `assertFormComponentActionShouldNotOpenUrlInNewTab()`:

```
use function Pest\Livewire\livewire;

it('links to the correct Filament sites', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertFormComponentActionHasUrl('customer_id', 'filament', 'https://filamentphp.com/')
    ->assertFormComponentActionDoesNotHaveUrl('customer_id', 'filament',
    'https://github.com/filamentphp/filament')
    ->assertFormComponentActionShouldOpenUrlInNewTab('customer_id', 'filament')
    ->assertFormComponentActionShouldNotOpenUrlInNewTab('customer_id', 'github');
});
```

Upgrade Guide

If you see anything missing from this guide, please do not hesitate to [make a pull request](#) to our repository! Any help is appreciated!

New requirements

- Laravel v10.0+
- Livewire v3.0+

Please upgrade Filament before upgrading to Livewire v3. Instructions on how to upgrade Livewire can be found [here](#).

Upgrading automatically

The easiest way to upgrade your app is to run the automated upgrade script. This script will automatically upgrade your application to the latest version of Filament, and make changes to your code which handle most breaking changes.

```
composer require filament/upgrade:"^3.2" -W --dev
vendor/bin/filament-v3
```

Make sure to carefully follow the instructions, and review the changes made by the script. You may need to make some manual changes to your code afterwards, but the script should handle most of the repetitive work for you.

Finally, you must run `php artisan filament:install` to finalize the Filament v3 installation. This command must be run for all new Filament projects.

You can now `composer remove filament/upgrade` as you don't need it anymore.

Some plugins you're using may not be available in v3 just yet. You could temporarily remove them from your `composer.json` file until they've been upgraded, replace them with a similar plugins that are v3-compatible, wait for the plugins to be upgraded before upgrading your app, or even write PRs to help the authors upgrade them.

Upgrading manually

After upgrading the dependency via Composer, you should execute `php artisan filament:upgrade` in order to clear any Laravel caches and publish the new frontend assets.

High-impact changes

Config file renamed and combined with other Filament packages

Only one config file is now used for all Filament packages. Most configuration has been moved into other parts of the codebase, and little remains. You should use the v3 documentation as a reference when replace the configuration options you did modify. To publish the new configuration file and remove the old one, run:

```
php artisan vendor:publish --tag=filament-config --force
rm config/forms.php
```

`FORMS_FILESYSTEM_DRIVER`.env variable

The `FORMS_FILESYSTEM_DRIVER`.env variable has been renamed to `FILAMENT_FILESYSTEM_DISK`. This is to make it more consistent with Laravel, as Laravel v9 introduced this change as well. Please ensure that you update your .env files accordingly, and don't forget production!

New `@filamentScripts` and `@filamentStyles` Blade directives

The `@filamentScripts` and `@filamentStyles` Blade directives must be added to your Blade layout file/s. Since Livewire v3 no longer uses similar directives, you can replace `@livewireScripts` with `@filamentScripts` and `@livewireStyles` with `@filamentStyles`.

CSS file removed

The CSS file for form components, `module.esm.css`, has been removed. Check `resources/css/app.css`. That CSS is now automatically loaded by `@filamentStyles`.

JavaScript files removed

You no longer need to import the `FormsAlpinePlugin` in your JavaScript files. Alpine plugins are now automatically loaded by `@filamentScripts`.

Heroicons have been updated to v2

The Heroicons library has been updated to v2. This means that any icons you use in your app may have changed names. You can find a list of changes [here](#).

Medium-impact changes

Date-time pickers

The date-time picker form field now uses the browser's native date picker by default. It usually has a better UX than the old date picker, but you may notice features missing, bad browser compatibility, or behavioral bugs. If you want to revert to the old date picker, you can use the `native(false)` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->native(false)
```

Secondary color

Filament v2 had a `secondary` color for many components which was gray. All references to `secondary` should be replaced with `gray` to preserve the same appearance. This frees `secondary` to be registered to a new custom color of your choice.

`$get` and `$set` closure parameters

`$get` and `$set` parameters now use a type of either `\Filament\Forms\Get` or `\Filament\Forms\Set` instead of `\Closure`. This allows for better IDE autocomplete support of each function's parameters.

An easy way to upgrade your code quickly is to find and replace:

- `Closure $get` to `\Filament\Forms\Get $get`
- `Closure $set` to `\Filament\Forms\Set $set`

`TextInput` masks now use Alpine.js' masking package

Filament v2 had a fluent mask object syntax for managing input masks. In v3, you can use Alpine.js's masking syntax instead. Please see the [input masking documentation](#) for more information.

Low-impact changes

Rule modification callback parameter renamed

The parameter for modifying rule objects has been renamed to `modifyRuleUsing()`, affecting:

- `exists()`
- `unique()`

Chapter 3

Table Builder

Installation

The Table Builder package is pre-installed with the [Panel Builder](#). This guide is for using the Table Builder in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

Require the Table Builder package using Composer:

```
composer require filament/tables:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --tables
npm install
npm run dev
```

Existing Laravel projects

Run the following command to install the Table Builder assets:

```
php artisan filament:install --tables
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament `preset` (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Upgrading from Filament v2? Please review the [upgrade guide](#).

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post/autoload/dump` hook, you can run the command yourself:

```
composer update  
php artisan filament:upgrade
```

Getting Started

Overview

Filament's Table Builder package allows you to [add an interactive datatable to any Livewire component](#). It's also used within other Filament packages, such as the [Panel Builder](#) for displaying [resources](#) and [relation managers](#), as well as for the [table widget](#). Learning the features of the Table Builder will be incredibly time-saving when both building your own custom Livewire tables and using Filament's other packages.

This guide will walk you through the basics of building tables with Filament's table package. If you're planning to add a new table to your own Livewire component, you should [do that first](#) and then come back. If you're adding a table to an [app resource](#), or another Filament package, you're ready to go!

Defining table columns

The basis of any table is rows and columns. Filament uses Eloquent to get the data for rows in the table, and you are responsible for defining the columns that are used in that row.

Filament includes many column types prebuilt for you, and you can [view a full list here](#). You can even [create your own custom column types](#) to display data in whatever way you need.

Columns are stored in an array, as objects within the `$table->columns()` method:

```
use Filament\Tables\Columns\IconColumn;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            TextColumn::make('title'),
            TextColumn::make('slug'),
            IconColumn::make('is_featured')
                ->boolean(),
        ]);
}
```

Title	Slug	Is featured
What is Filament?	what-is-filament	✓
Top 5 best features of Filament	top-5-features	✗
Tips for building a great Filament plugin	plugin-tips	✓
Customizing Filament's UI with a theme	theme-guide	✗
New Filament plugins in August	new-plugins-august	✗

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

In this example, there are 3 columns in the table. The first two display text - the title and slug of each row in the table. The third column displays an icon, either a green check or a red cross depending on if the row is featured or not.

Making columns sortable and searchable

You can easily modify columns by chaining methods onto them. For example, you can make a column searchable using the `searchable()` method. Now, there will be a search field in the table, and you will be able to filter rows by the value of that column:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->searchable()
```

<table border="1"> <tr> <td colspan="3"></td><td>Search</td></tr> </table>						Search
			Search			
Title	Slug	Is featured				
What is Filament?	what-is-filament	✓				
Top 5 best features of Filament	top-5-features	✗				
Tips for building a great Filament plugin	plugin-tips	✓				
Customizing Filament's UI with a theme	theme-guide	✗				
New Filament plugins in August	new-plugins-august	✗				

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

You can make multiple columns searchable, and Filament will be able to search for matches within any of them, all at once.

You can also make a column sortable using the `sortable()` method. This will add a sort button to the column header, and clicking it will sort the table by that column:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->sortable()
```

Title	Slug	Is featured
What is Filament?	what-is-filament	✓
Top 5 best features of Filament	top-5-features	✗
Tips for building a great Filament plugin	plugin-tips	✓
Customizing Filament's UI with a theme	theme-guide	✗
New Filament plugins in August	new-plugins-august	✗

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

Accessing related data from columns

You can also display data in a column that belongs to a relationship. For example, if you have a `Post` model that belongs to a `User` model (the author of the post), you can display the user's name in the table:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('author.name')
```

Title	Slug	Is featured	Author
What is Filament?	what-is-filament	✓	Dan Harrin
Top 5 best features of Filament	top-5-features	✗	Ryan Chandler
Tips for building a great Filament plugin	plugin-tips	✓	Zep Fietje
Customizing Filament's UI with a theme	theme-guide	✗	Dennis Koch
New Filament plugins in August	new-plugins-august	✗	Adam Weston

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

In this case, Filament will search for an `author` relationship on the `Post` model, and then display the `name` attribute of that relationship. We call this "dot notation" - you can use it to display any attribute of any relationship, even nested

distant relationships. Filament uses this dot notation to eager-load the results of that relationship for you.

Defining table filters

As well as making columns `searchable()`, you can allow the users to filter rows in the table in other ways. We call these components "filters", and they are defined in the `$table->filters()` method:

```
use Filament\Tables\Filters\Filter;
use Filament\Tables\Filters>SelectFilter;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters([
            Filter::make('is_featured')
                ->query(fn (Builder $query) => $query->where('is_featured', true)),
            SelectFilter::make('status')
                ->options([
                    'draft' => 'Draft',
                    'reviewing' => 'Reviewing',
                    'published' => 'Published',
                ]),
        ]);
}
```

The screenshot shows a Filament table interface. On the left, there's a table with columns 'Title' and 'Slug'. The table contains five rows of data. To the right of the table is a 'Filters' dropdown menu. The menu has two sections: 'Is featured' (with a checkbox) and 'Status' (with a dropdown menu set to 'All'). There are also 'Reset' and 'Merge' buttons. At the bottom of the table area, there are pagination controls showing 'Showing 1 to 5 of 50 results', 'Per page' (set to 5), and a page navigation bar with numbers 1 through 10.

Title	Slug
What is Filament?	what-is-filament
Top 5 best features of Filament	top-5-features
Tips for building a great Filament plugin	plugin-tips
Customizing Filament's UI with a theme	theme-guide
New Filament plugins in August	new-plugins-august

In this example, we have defined 2 table filters. On the table, there is now a "filter" icon button in the top corner. Clicking it will open a dropdown with the 2 filters we have defined.

The first filter is rendered as a checkbox. When it's checked, only featured rows in the table will be displayed. When it's unchecked, all rows will be displayed.

The second filter is rendered as a select dropdown. When a user selects an option, only rows with that status will be displayed. When no option is selected, all rows will be displayed.

It's possible to define as many filters as you need, and use any component from the [Form Builder package](#) to create a UI. For example, you could create [a custom date range filter](#).

Defining table actions

Filament's tables can use [Actions](#). They are buttons that can be added to the [end of any table row](#), or even in the [header](#) of a table. For instance, you may want an action to "create" a new record in the header, and then "edit" and "delete" actions on each row. [Bulk actions](#) can be used to execute code when records in the table are selected.

```
use App\Models\Post;
use Filament\Tables\Actions\Action;
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Actions>DeleteBulkAction;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->actions([
            Action::make('feature')
                ->action(function (Post $record) {
                    $record->is_featured = true;
                    $record->save();
                })
                ->hidden(fn (Post $record): bool => $record->is_featured),
            Action::make('unfeature')
                ->action(function (Post $record) {
                    $record->is_featured = false;
                    $record->save();
                })
                ->visible(fn (Post $record): bool => $record->is_featured),
        ])
        ->bulkActions([
            BulkActionGroup::make([
                DeleteBulkAction::make(),
            ]),
        ]);
}
```

Title	Slug	Is featured	Author
What is Filament?	what-is-filament	✓	Dan Harrin
Top 5 best features of Filament	top-5-features	✗	Ryan Chandler
Tips for building a great Filament plugin	plugin-tips	✓	Zep Fietje
Customizing Filament's UI with a theme	theme-guide	✗	Dennis Koch
New Filament plugins in August	new-plugins-august	✗	Adam Weston

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

In this example, we define 2 actions for table rows. The first action is a "feature" action. When clicked, it will set the `is_featured` attribute on the record to `true` - which is written within the `action()` method. Using the `hidden()` method, the action will be hidden if the record is already featured. The second action is an "unfeature" action. When clicked, it will set the `is_featured` attribute on the record to `false`. Using the `visible()` method, the action will be hidden if the record is not featured.

We also define a bulk action. When bulk actions are defined, each row in the table will have a checkbox. This bulk action is [built-in to Filament](#), and it will delete all selected records. However, you can [write your own custom bulk actions](#) easily too.

Bulk actions		Author	
<input checked="" type="checkbox"/> Delete selected		Select all 50 Deselect all	
<input checked="" type="checkbox"/>	Title	Dan Harrin	Unfeature
<input checked="" type="checkbox"/>	What is Filament?	Ryan Chandler	Feature
<input checked="" type="checkbox"/>	Top 5 best features of Filament	Zep Fietje	Unfeature
<input checked="" type="checkbox"/>	Tips for building a great Filament pl...	Dennis Koch	Feature
<input checked="" type="checkbox"/>	Customizing Filament's UI with a theme	Adam Weston	Feature
<input checked="" type="checkbox"/>	New Filament plugins in August		

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Actions can also open modals to request confirmation from the user, as well as render forms inside to collect extra data. It's a good idea to read the [Actions documentation](#) to learn more about their extensive capabilities throughout Filament.

Next steps with the Table Builder package

Now you've finished reading this guide, where to next? Here are some suggestions:

- [Explore the available columns to display data in your table.](#)

- [Deep dive into table actions and start using modals.](#)
- [Discover how to build complex, responsive table layouts without touching CSS.](#)
- [Add summaries to your tables, which give an overview of the data inside them.](#)
- [Find out about all advanced techniques that you can customize tables to your needs.](#)
- [Write automated tests for your tables using our suite of helper methods.](#)

Columns

Getting Started

Overview

Column classes can be found in the `Filament\Tables\Columns` namespace. You can put them inside the `$table->columns()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ]);
}
```

Columns may be created using the static `make()` method, passing its unique name. The name of the column should correspond to a column or accessor on your model. You may use "dot notation" to access columns within relationships.

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')

TextColumn::make('author.name')
```

Available columns

Filament ships with two main types of columns - static and editable.

Static columns display data to the user:

- [Text column](#)
- [Icon column](#)
- [Image column](#)
- [Color column](#)

Editable columns allow the user to update data in the database without leaving the table:

- [Select column](#)
- [Toggle column](#)
- [Text input column](#)
- [Checkbox column](#)

You may also [create your own custom columns](#) to display data however you wish.

Setting a label

By default, the label of the column, which is displayed in the header of the table, is generated from the name of the column. You may customize this using the `label()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->label('Post title')
```

Optionally, you can have the label automatically translated [using Laravel's localization features](#) with the `translateLabel()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->translateLabel() // Equivalent to `label(__('Title'))`
```

Sorting

Columns may be sortable, by clicking on the column label. To make a column sortable, you must use the `sortable()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
->sortable()
```

Name	Email address	Verified
Dan Harrin	dan@filamentphp.com	✓
Ryan Chandler	ryan@filamentphp.com	✗
Zep Fietje	zep@filamentphp.com	✗
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓

Showing 1 to 5 of 51 results

Per page: 5 ▾

1 2 3 4 ... 10 11 >

If you're using an accessor column, you may pass `sortable()` an array of database columns to sort by:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('full_name')
->sortable(['first_name', 'last_name'])
```

You may customize how the sorting is applied to the Eloquent query using a callback:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('full_name')
    ->sortable(query: function (Builder $query, string $direction): Builder {
        return $query
            ->orderBy('last_name', $direction)
            ->orderBy('first_name', $direction);
    })
}
```

Sorting by default

You may choose to sort a table by default if no other sort is applied. You can use the `defaultSort()` method for this:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->defaultSort('stock', 'desc');
}
```

Persist sort in session

To persist the sorting in the user's session, use the `persistSortInSession()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->persistSortInSession();
}
```

Setting a default sort option label

To set a default sort option label, use the `defaultSortOptionLabel()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->defaultSortOptionLabel('Date');
}
```

Searching

Columns may be searchable by using the text input field in the top right of the table. To make a column searchable, you must use the `searchable()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->searchable()
```

Name	Email address	Verified
Dan Harrin	dan@filamentphp.com	✓
Ryan Chandler	ryan@filamentphp.com	✗
Zep Fietje	zep@filamentphp.com	✗
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓

Showing 1 to 5 of 51 results

Per page: 5 ▾

1 2 3 4 ... 10 11 >

If you're using an accessor column, you may pass `searchable()` an array of database columns to search within:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('full_name')
    ->searchable(['first_name', 'last_name'])
```

You may customize how the search is applied to the Eloquent query using a callback:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('full_name')
    ->searchable(query: function (Builder $query, string $search): Builder {
        return $query
            ->where('first_name', 'like', "%{$search}%")
            ->orWhere('last_name', 'like', "%{$search}%");
    })
}
```

Customizing the table search field placeholder

You may customize the placeholder in the search field using the `searchPlaceholder()` method on the `$table`:

```
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->searchPlaceholder('Search (ID, Name)');
}
```

Searching individually

You can choose to enable a per-column search input field using the `isIndividual` parameter:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->searchable(isIndividual: true)
```

Name	Email address	Verified
<input type="text" value="Search"/>	<input type="text" value="Search"/>	
Dan Harrin	dan@filamentphp.com	✓
Ryan Chandler	ryan@filamentphp.com	✗
Zep Fietje	zep@filamentphp.com	✗
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓
Showing 1 to 5 of 51 results		Per page 5 <
		1 2 3 4 ... 10 11 >

If you use the `isIndividual` parameter, you may still search that column using the main "global" search input field for the entire table.

To disable that functionality while still preserving the individual search functionality, you need the `isGlobal` parameter:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->searchable(isIndividual: true, isGlobal: false)
```

You may optionally persist the searches in the query string:

```
use Livewire\Attributes\Url;

/**
 * @var array<string, string | array<string, string | null> | null>
 */
#[Url]
public array $tableColumnSearches = [];
```

Customizing the table search debounce

You may customize the debounce time in all table search fields using the `searchDebounce()` method on the `$table`. By default it is set to `500ms`:

```
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->searchDebounce('750ms');
}
```

Searching when the input is blurred

Instead of automatically reloading the table contents while the user is typing their search, which is affected by the `debounce` of the search field, you may change the behavior so that the table is only searched when the user blurs the input (tabs or clicks out of it), using the `searchOnBlur()` method:

```
use Filament\Tables\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->searchOnBlur();
}
```

Persist search in session

To persist the table or individual column search in the user's session, use the `persistSearchInSession()` or `persistColumnSearchInSession()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->persistSearchInSession()
        ->persistColumnSearchesInSession();
}
```

Column actions and URLs

When a cell is clicked, you may run an "action", or open a URL.

Running actions

To run an action, you may use the `action()` method, passing a callback or the name of a Livewire method to run. Each method accepts a `$record` parameter which you may use to customize the behavior of the action:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->action(function (Post $record): void {
        $this->dispatch('open-post-edit-modal', post: $record->getKey());
    })
```

Action modals

You may open [action modals](#) by passing in an `Action` object to the `action()` method:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->action(
        Action::make('select')
            ->requiresConfirmation()
            ->action(function (Post $record): void {
                $this->dispatch('select-post', post: $record->getKey());
            }),
    )
```

Action objects passed into the `action()` method must have a unique name to distinguish it from other actions within the table.

Opening URLs

To open a URL, you may use the `url()` method, passing a callback or static URL to open. Callbacks accept a `$record` parameter which you may use to customize the URL:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->url(fn (Post $record): string => route('posts.edit', ['post' => $record]))
```

You may also choose to open the URL in a new tab:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->url(fn (Post $record): string => route('posts.edit', ['post' => $record]))
->openUrlInNewTab()
```

Setting a default value

To set a default value for columns with an empty state, you may use the `default()` method. This method will treat the default state as if it were real, so columns like `image` or `color` will display the default image or color.

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->default('No description.')
```

Adding placeholder text if a column is empty

Sometimes you may want to display placeholder text for columns with an empty state, which is styled as a lighter gray text. This differs from the [default value](#), as the placeholder is always text and not treated as if it were real state.

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->placeholder('No description.')
```

Title	Description
What is Filament?	Find out what Filament is and how it can help you build your next project.
Top 5 best features of Filament	No description.
Tips for building a great Filament plugin	Learn how to build a great Filament plugin and get it featured in the official plugin directory.
Customizing Filament's UI with a theme	No description.
New Filament plugins in August	Discover the latest Filament plugins that were released in August.

Showing 1 to 5 of 50 results

Per page

5 ▾

1
2
3
4
...
9
10
>

Hiding columns

To hide a column conditionally, you may use the `hidden()` and `visible()` methods, whichever you prefer:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('role')
    ->hidden(! auth()->user()->isAdmin())
// or
TextColumn::make('role')
    ->visible(auth()->user()->isAdmin())
```

Toggling column visibility

Users may hide or show columns themselves in the table. To make a column toggleable, use the `toggleable()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->toggleable()
```

The screenshot shows a table component from the Filament UI library. The table has two columns: 'Name' and 'Email address'. On the right side of the table, there is a 'Columns' dropdown menu. It contains two items: 'Email address' with a checked checkbox and 'Verified' with an unchecked checkbox. Below the table, there is a message 'Showing 1 to 5 of 51 results' and a pagination control 'Per page 5 < >'. The page number 1 is highlighted in orange, indicating it is the current page.

Name	Email address	
Dan Harrin	dan@filamentphp.com	
Ryan Chandler	ryan@filamentphp.com	
Zep Fietje	zep@filamentphp.com	✖
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓

Making toggleable columns hidden by default

By default, toggleable columns are visible. To make them hidden instead:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('id')
    ->toggleable(isToggledHiddenByDefault: true)
```

Customizing the toggle columns dropdown trigger action

To customize the toggle dropdown trigger button, you may use the `toggleColumnsTriggerAction()` method, passing a closure that returns an action. All methods that are available to [customize action trigger buttons](#) can be used:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->toggleColumnsTriggerAction(
            fn (Action $action) => $action
                ->button()
                ->label('Toggle columns'),
            );
}
```

Calculated state

Sometimes you need to calculate the state of a column, instead of directly reading it from a database column.

By passing a callback function to the `state()` method, you can customize the returned state for that column based on the `$record`:

```
use App\Models\Order;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('amount_including_vat')
    ->state(function (Order $record): float {
        return $record->amount * (1 + $record->vat_rate);
    })
```

Tooltips

You may specify a tooltip to display when you hover over a cell:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->tooltip('Title')
```

Name	Email address	Verified
Dan Harrin	dan@filamentphp.com	✓
Ryan Chandler	ryan@filamentphp.com	✗
Zep Fietje	zep@filamentphp.com	✗ Aug 1, 2023
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

This method also accepts a closure that can access the current table record:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Model;

TextColumn::make('title')
    ->tooltip(fn (Model $record): string => "By {$record->author->name}")
```

Horizontally aligning column content

Table columns are aligned to the start (left in LTR interfaces or right in RTL interfaces) by default. You may change the alignment using the `alignment()` method, and passing it `Alignment::Start`, `Alignment::Center`, `Alignment::End` or `Alignment::Justify` options:

```
use Filament\Support\Enums\Alignment;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->alignment(Alignment::End)
```

Name	Email address	Verified
Dan Harrin	dan@filamentphp.com	✓
Ryan Chandler	ryan@filamentphp.com	✗
Zep Fietje	zep@filamentphp.com	✗
Dennis Koch	dennis@filamentphp.com	✓
Adam Weston	adam@filamentphp.com	✓

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

Alternatively, you may use shorthand methods like `alignEnd()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->alignEnd()
```

Vertically aligning column content

Table column content is vertically centered by default. You may change the vertical alignment using the `verticalAlignment()` method, and passing it `VerticalAlignment::Start`, `VerticalAlignment::Center` or `VerticalAlignment::End` options:

```
use Filament\Support\Enums\VerticalAlignment;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->verticalAlignment(VerticalAlignment::Start)
```

Name	Email addresses	Verified
Dan Harrin	dan@filamentphp.com dan@filament.dev	✓
Ryan Chandler	ryan@filamentphp.com ryan@filament.dev	✗
Zep Fietje	zep@filamentphp.com zep@filament.dev	✗
Dennis Koch	dennis@filamentphp.com dennis@filament.dev	✓
Adam Weston	adam@filamentphp.com adam@filament.dev	✓

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

Alternatively, you may use shorthand methods like `verticallyAlignStart()`:

```
use Filament\Support\Enums\VerticalAlignment;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->verticallyAlignStart()
```

Allowing column headers to wrap

By default, column headers will not wrap onto multiple lines, if they need more space. You may allow them to wrap using the `wrapHeader()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
->wrapHeader()
```

Controlling the width of columns

By default, columns will take up as much space as they need. You may allow some columns to consume more space than others by using the `grow()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
->grow()
```

Alternatively, you can define a width for the column, which is passed to the header cell using the `style` attribute, so you can use any valid CSS value:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_paid')
->label('Paid')
->boolean()
->width('1%')
```

Grouping columns

You group multiple columns together underneath a single heading using a `ColumnGroup` object:

```
use Filament\Tables\Columns\ColumnGroup;
use Filament\Tables\Columns\IconColumn;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            TextColumn::make('title'),
            TextColumn::make('slug'),
            ColumnGroup::make('Visibility', [
                TextColumn::make('status'),
                IconColumn::make('is_featured'),
            ]),
            TextColumn::make('author.name'),
        ]);
}
```

The first argument is the label of the group, and the second is an array of column objects that belong to that group.

Title	Slug	Visibility		
		Status	Is featured	Author
What is Filament?	what-is-filament	published	✓	Dan Harrin
Top 5 best features of Filament	top-5-features	reviewing	✗	Ryan Chandler
Tips for building a great Filament plugin	plugin-tips	draft	✓	Zep Fietje
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	Dennis Koch
New Filament plugins in August	new-plugins-august	published	✗	Adam Weston
Showing 1 to 5 of 50 results		Per page	5	1 2 3 4 ... 9 10 >

You can also control the group header `alignment` and `wrapping` on the `ColumnGroup` object. To improve the multi-line fluency of the API, you can chain the `columns()` onto the object instead of passing it as the second argument:

```
use Filament\Support\Enums\Alignment;
use Filament\Tables\Columns\ColumnGroup;

ColumnGroup::make('Website visibility')
    ->columns([
        // ...
    ])
    ->alignment(Alignment::Center)
    ->wrapHeader()
```

Custom attributes

The HTML of columns can be customized, by passing an array of `extraAttributes()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('slug')
    ->extraAttributes(['class' => 'bg-gray-200'])
```

These get merged onto the outer `<div>` element of each cell in that column.

Global settings

If you wish to change the default behavior of all columns globally, then you can call the static `configureUsing()` method inside a service provider's `boot()` method, to which you pass a Closure to modify the columns using. For example, if you wish to make all columns `searchable()` and `toggleable()`, you can do it like so:

```
use Filament\Tables\Columns\Column;

Column::configureUsing(function (Column $column): void {
    $column
        ->toggleable()
        ->searchable();
});
```

Additionally, you can call this code on specific column types as well:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::configureUsing(function (TextColumn $column): void {
    $column
        ->toggleable()
        ->searchable();
});
```

Of course, you are still able to overwrite this on each column individually:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')
    ->toggleable(false)
```

Text

Overview

Text columns display simple text from your database:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
```

Title
What is Filament?
Top 5 best features of Filament
Tips for building a great Filament plugin
Customizing Filament's UI with a theme
New Filament plugins in August
Showing 1 to 5 of 50 results
<div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

Displaying as a "badge"

By default, the text is quite plain and has no background color. You can make it appear as a "badge" instead using the `badge()` method. A great use case for this is with statuses, where you may want to display a badge with a `color` that matches the status:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')
    ->badge()
    ->color(fn (string $state): string => match ($state) {
        'draft' => 'gray',
        'reviewing' => 'warning',
        'published' => 'success',
        'rejected' => 'danger',
    })
```

Title	Status
What is Filament?	published
Top 5 best features of Filament	reviewing
Tips for building a great Filament plugin	draft
Customizing Filament's UI with a theme	reviewing
New Filament plugins in August	published
Showing 1 to 5 of 50 results	<div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

You may add other things to the badge, like an [icon](#).

Displaying a description

Descriptions may be used to easily render additional text above or below the column contents.

You can display a description below the contents of a text column using the `description()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->description(fn (Post $record): string => $record->description)
```

Title	
What is Filament? Find out what Filament is and how it can help you build your next project.	
Top 5 best features of Filament Discover the top 5 best features of Filament and how they can help you build your next project.	
Tips for building a great Filament plugin Learn how to build a great Filament plugin and get it featured in the official plugin directory.	
Customizing Filament's UI with a theme Learn how to customize Filament's UI with a theme and make it your own.	
New Filament plugins in August Discover the latest Filament plugins that were released in August.	
Showing 1 to 5 of 50 results	<div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

By default, the description is displayed below the main text, but you can move it above using the second parameter:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->description(fn (Post $record): string => $record->description, position: 'above')
```

Title
Find out what Filament is and how it can help you build your next project. What is Filament?
Discover the top 5 best features of Filament and how they can help you build your next project. Top 5 best features of Filament
Learn how to build a great Filament plugin and get it featured in the official plugin directory. Tips for building a great Filament plugin
Learn how to customize Filament's UI with a theme and make it your own. Customizing Filament's UI with a theme
Discover the latest Filament plugins that were released in August. New Filament plugins in August
Showing 1 to 5 of 50 results
Per page 5 1 2 3 4 ... 9 10 >

Date formatting

You may use the `date()` and `dateTime()` methods to format the column's state using [PHP date formatting tokens](#):

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('created_at')
    ->dateTime()
```

You may use the `since()` method to format the column's state using [Carbon's `diffForHumans\(\)`](#):

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('created_at')
    ->since()
```

Additionally, you can use the `dateTooltip()`, `dateTimeTooltip()` or `timeTooltip()` method to display a formatted date in a tooltip, often to provide extra information:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('created_at')
    ->since()
    ->dateTimeTooltip()
```

Number formatting

The `numeric()` method allows you to format an entry as a number:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('stock')
->numeric()
```

If you would like to customize the number of decimal places used to format the number with, you can use the `decimalPlaces` argument:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('stock')
->numeric(decimalPlaces: 0)
```

By default, your app's locale will be used to format the number suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('stock')
->numeric(locale: 'nl')
```

Alternatively, you can set the default locale used across your app using the `Table::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Tables\Table;

Table::$defaultNumberLocale = 'nl';
```

Currency formatting

The `money()` method allows you to easily format monetary values, in any currency:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->money('EUR')
```

There is also a `divideBy` argument for `money()` that allows you to divide the original value by a number before formatting it. This could be useful if your database stores the price in cents, for example:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->money('EUR', divideBy: 100)
```

By default, your app's locale will be used to format the money suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
    ->money('EUR', locale: 'nl')
```

Alternatively, you can set the default locale used across your app using the `Table::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Tables\Table;

Table::$defaultNumberLocale = 'nl';
```

Limits

You may `limit()` the length of the cell's value:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
    ->limit(50)
```

You may also reuse the value that is being passed to `limit()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
    ->limit(50)
    ->tooltip(function (TextColumn $column): ?string {
        $state = $column->getState();

        if (strlen($state) <= $column->getCharacterLimit()) {
            return null;
        }

        // Only render the tooltip if the column content exceeds the length limit.
        return $state;
    })
```

Limits

You may limit the number of `words()` displayed in the cell:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
    ->words(10)
```

Limits

You may want to limit text to a specific number of lines instead of limiting it to a fixed length. Clamping text to a number of lines is useful in responsive interfaces where you want to ensure a consistent experience across all screen sizes. This can

be achieved using the `lineClamp()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->lineClamp(2)
```

Adding a prefix or suffix

You may add a prefix or suffix to the cell's value:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('domain')
->prefix('https://')
->suffix('.com')
```

Wrapping content

If you'd like your column's content to wrap if it's too long, you may use the `wrap()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->wrap()
```

Listing multiple values

By default, if there are multiple values inside your text column, they will be comma-separated. You may use the `listWithLineBreaks()` method to display them on new lines instead:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('authors.name')
->listWithLineBreaks()
```

Adding bullet points to the list

You may add a bullet point to each list item using the `bulleted()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('authors.name')
->listWithLineBreaks()
->bulleted()
```

Limiting the number of values in the list

You can limit the number of values in the list using the `limitList()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('authors.name')
->listWithLineBreaks()
->limitList(3)
```

Expanding the limited list

You can allow the limited items to be expanded and collapsed, using the `expandableLimitedList()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('authors.name')
->listWithLineBreaks()
->limitList(3)
->expandableLimitedList()
```

Please note that this is only a feature for `listWithLineBreaks()` or `bulleted()`, where each item is on its own line.

Using a list separator

If you want to "explode" a text string from your model into multiple list items, you can do so with the `separator()` method. This is useful for displaying comma-separated tags as badges, for example:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('tags')
->badge()
->separator(',')
```

Rendering HTML

If your column value is HTML, you may render it using `html()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->html()
```

If you use this method, then the HTML will be sanitized to remove any potentially unsafe content before it is rendered. If you'd like to opt out of this behavior, you can wrap the HTML in an `HtmlString` object by formatting it:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Support\HtmlString;

TextColumn::make('description')
->formatStateUsing(fn (string $state): HtmlString => new HtmlString($state))
```

Or, you can return a `view()` object from the `formatStateUsing()` method, which will also not be sanitized:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Contracts\View\View;

TextColumn::make('description')
->formatStateUsing(fn (string $state): View => view(
    'filament.tables.columns.description-entry-content',
    ['state' => $state],
))
)
```

Rendering Markdown as HTML

If your column contains Markdown, you may render it using `markdown()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')
->markdown()
```

Custom formatting

You may instead pass a custom formatting callback to `formatStateUsing()`, which accepts the `$state` of the cell, and optionally its `$record`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')
->formatStateUsing(fn (string $state): string => __("statuses.{${$state}}"))
```

Customizing the color

You may set a color for the text, either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')
->color('primary')
```

Title	Status
What is Filament?	published
Top 5 best features of Filament	reviewing
Tips for building a great Filament plugin	draft
Customizing Filament's UI with a theme	reviewing
New Filament plugins in August	published

Showing 1 to 5 of 50 results Per page: 5 ▾ 1 2 3 4 ... 9 10 >

Adding an icon

Text columns may also have an icon:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
->icon('heroicon-m-envelope')
```

Name	Email
Dan Harrin	✉ dan@filamentphp.com
Ryan Chandler	✉ ryan@filamentphp.com
Zep Fietje	✉ zep@filamentphp.com
Dennis Koch	✉ dennis@filamentphp.com
Adam Weston	✉ adam@filamentphp.com

Showing 1 to 5 of 51 results Per page: 5 ▾ 1 2 3 4 ... 10 11 >

You may set the position of an icon using `iconPosition()`:

```
use Filament\Support\Enums\IconPosition;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
->icon('heroicon-m-envelope')
->iconPosition(IconPosition::After) // `IconPosition::Before` or `IconPosition::After`
```

Name	Email
Dan Harrin	dan@filamentphp.com 
Ryan Chandler	ryan@filamentphp.com 
Zep Fietje	zep@filamentphp.com 
Dennis Koch	dennis@filamentphp.com 
Adam Weston	adam@filamentphp.com 

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

The icon color defaults to the text color, but you may customize the icon color separately using `iconColor()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->icon('heroicon-m-envelope')
    ->iconColor('primary')
```

Name	Email
Dan Harrin	 dan@filamentphp.com
Ryan Chandler	 ryan@filamentphp.com
Zep Fietje	 zep@filamentphp.com
Dennis Koch	 dennis@filamentphp.com
Adam Weston	 adam@filamentphp.com

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

Customizing the text size

Text columns have small font size by default, but you may change this to `TextColumnSize::ExtraSmall`, `TextColumnSize::Medium`, or `TextColumnSize::Large`.

For instance, you may make the text larger using `size(TextColumnSize::Large)`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->size(TextColumn\TextColumnSize::Large)
```

Title
What is Filament?
Top 5 best features of Filament
Tips for building a great Filament plugin
Customizing Filament's UI with a theme
New Filament plugins in August
<p>Showing 1 to 5 of 50 results</p> <div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

Customizing the font weight

Text columns have regular font weight by default, but you may change this to any of the following options:

`FontWeight::Thin`, `FontWeight::ExtraLight`, `FontWeight::Light`, `FontWeight::Medium`,
`FontWeight::SemiBold`, `FontWeight::Bold`, `FontWeight::ExtraBold` or `FontWeight::Black`.

For instance, you may make the font bold using `weight(FontWeight::Bold)`:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->weight(FontWeight::Bold)
```

Title
What is Filament?
Top 5 best features of Filament
Tips for building a great Filament plugin
Customizing Filament's UI with a theme
New Filament plugins in August
<p>Showing 1 to 5 of 50 results</p> <div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

Customizing the font family

You can change the text font family to any of the following options: `FontFamily::Sans`, `FontFamily::Serif` or `FontFamily::Mono`.

For instance, you may make the font mono using `fontFamily(FontFamily::Mono)`:

```
use Filament\Support\Enums\FontFamily;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->fontFamily(FontFamily::Mono)
```

Name	Email
Dan Harrin	dan@filamentphp.com
Ryan Chandler	ryan@filamentphp.com
Zep Fietje	zep@filamentphp.com
Dennis Koch	dennis@filamentphp.com
Adam Weston	adam@filamentphp.com

Showing 1 to 5 of 51 results

Per page
5 ▾
1 2 3 4 ... 10 11 >

Allowing the text to be copied to the clipboard

You may make the text copyable, such that clicking on the cell copies the text to the clipboard, and optionally specify a custom confirmation message and duration in milliseconds. This feature only works when SSL is enabled for the app.

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->copyable()
    ->copyMessage('Email address copied')
    ->copyMessageDuration(1500)
```

Name	Email
Dan Harrin	dan@filamentphp.com
Ryan Chandler	ryan@filamentphp.com
Zep Fietje	zep@filamentphp.com
Dennis Koch	dennis@filamentphp.com
Adam Weston	adam@filamentphp.com

Showing 1 to 5 of 51 results

Per page
5 ▾
1 2 3 4 ... 10 11 >

Customizing the text that is copied to the clipboard

You can customize the text that gets copied to the clipboard using the `copyableState()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('url')
->copyable()
->copyableState(fn (string $state): string => "URL: {$state}")
```

In this function, you can access the whole table row with `$record`:

```
use App\Models\Post;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('url')
->copyable()
->copyableState(fn (Post $record): string => "URL: {$record->url}")
```

Displaying the row index

You may want a column to contain the number of the current row in the table:

```
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Contracts\HasTable;

TextColumn::make('index')->state(
    static function (HasTable $livewire, stdClass $rowLoop): string {
        return (string) (
            $rowLoop->iteration +
            ($livewire->getTableRecordsPerPage() * (
                $livewire->getTablePage() - 1
            )));
    });
),
```

As `$rowLoop` is Laravel Blade's `$loop` object, you can reference all other `$loop` properties.

As a shortcut, you may use the `rowIndex()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('index')
->rowIndex()
```

To start counting from 0 instead of 1, use `isFromZero: true`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('index')
->rowIndex(isFromZero: true)
```

Icon

Overview

Icon columns render an `icon` representing their contents:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('status')
->icon(fn (string $state): string => match ($state) {
    'draft' => 'heroicon-o-pencil',
    'reviewing' => 'heroicon-o-clock',
    'published' => 'heroicon-o-check-circle',
})
```

In the function, `$state` is the value of the column, and `$record` can be used to access the underlying Eloquent record.

Title	Status
What is Filament?	✓
Top 5 best features of Filament	🕒
Tips for building a great Filament plugin	✍
Customizing Filament's UI with a theme	🕒
New Filament plugins in August	✓
Showing 1 to 5 of 50 results	<div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

Customizing the color

Icon columns may also have a set of icon colors, using the same syntax. They may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('status')
->color(fn (string $state): string => match ($state) {
    'draft' => 'info',
    'reviewing' => 'warning',
    'published' => 'success',
    default => 'gray',
})
```

In the function, `$state` is the value of the column, and `$record` can be used to access the underlying Eloquent record.

Title	Status
What is Filament?	✓
Top 5 best features of Filament	🕒
Tips for building a great Filament plugin	📝
Customizing Filament's UI with a theme	🕒
New Filament plugins in August	✓

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

Customizing the size

The default icon size is `IconColumnSize::Large`, but you may customize the size to be either

`IconColumnSize::ExtraSmall`, `IconColumnSize::Small`, `IconColumnSize::Medium`,
`IconColumnSize::ExtraLarge` or `IconColumnSize::TwoExtraLarge`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('status')
->size(IconColumn\IconColumnSize::Medium)
```

Title	Status
What is Filament?	✓
Top 5 best features of Filament	🕒
Tips for building a great Filament plugin	📝
Customizing Filament's UI with a theme	🕒
New Filament plugins in August	✓

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

Handling booleans

Icon columns can display a check or cross icon based on the contents of the database column, either true or false, using the `boolean()` method:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->boolean()
```

If this column in the model class is already cast as a `bool` or `boolean`, Filament is able to detect this, and you do not need to use `boolean()` manually.

Title	Is featured
What is Filament?	✓
Top 5 best features of Filament	✗
Tips for building a great Filament plugin	✓
Customizing Filament's UI with a theme	✗
New Filament plugins in August	✗

Showing 1 to 5 of 50 results Per page: 5 ▾ 1 2 3 4 ... 9 10 >

Customizing the boolean icons

You may customize the icon representing each state. Icons are the name of a Blade component present. By default, [Heroicons](#) are installed:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->boolean()
->trueIcon('heroicon-o-check-badge')
->falseIcon('heroicon-o-x-mark')
```

Title	Is featured
What is Filament?	✓
Top 5 best features of Filament	✗
Tips for building a great Filament plugin	✓
Customizing Filament's UI with a theme	✗
New Filament plugins in August	✗

Showing 1 to 5 of 50 results Per page: 5 ▾ 1 2 3 4 ... 9 10 >

Customizing the boolean colors

You may customize the icon color representing each state. These may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->boolean()
->trueColor('info')
->falseColor('warning')
```

Title	Is featured
What is Filament?	✓
Top 5 best features of Filament	✗
Tips for building a great Filament plugin	✓
Customizing Filament's UI with a theme	✗
New Filament plugins in August	✗

Showing 1 to 5 of 50 results Per page: 5 ▾ 1 2 3 4 ... 9 10 >

Wrapping multiple icons

When displaying multiple icons, they can be set to wrap if they can't fit on one line, using `wrap()`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('icon')
->wrap()
```

Note: the "width" for wrapping is affected by the column label, so you may need to use a shorter or hidden label to wrap more tightly.

Image

Overview

Images can be easily displayed within your table:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('avatar')
```

The column in the database must contain the path to the image, relative to the root directory of its storage disk.

Name	Avatar
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	

Showing 1 to 5 of 51 results

Per page 5 ▾
 1 2 3 4 ... 10 11 >

Managing the image disk

By default, the `public` disk will be used to retrieve images. You may pass a custom disk name to the `disk()` method:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('header_image')
->disk('s3')
```

Private images

Filament can generate temporary URLs to render private images, you may set the `visibility()` to `private`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('header_image')
->visibility('private')
```

Customizing the size

You may customize the image size by passing a `width()` and `height()`, or both with `size()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('header_image')
->width(200)

ImageColumn::make('header_image')
->height(50)

ImageColumn::make('author.avatar')
->size(40)
```

Square image

You may display the image using a 1:1 aspect ratio:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('avatar')
->square()
```

Name	Avatar
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	

Showing 1 to 5 of 51 results Per page: 5 ▾ 1 2 3 4 ... 10 11 >

Circular image

You may make the image fully rounded, which is useful for rendering avatars:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('avatar')
->circular()
```

Name	Avatar
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	

Showing 1 to 5 of 51 results Per page: 5 < 1 2 3 4 ... 10 11 >

Adding a default image URL

You can display a placeholder image if one doesn't exist yet, by passing a URL to the `defaultImageUrl()` method:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('avatar')
    ->defaultImageUrl(url('/images/placeholder.png'))
```

Stacking images

You may display multiple images as a stack of overlapping images by using `stacked()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
```

Name	Colleagues
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	
Showing 1 to 5 of 51 results Per page <input type="button" value="5"/> <input type="button" value="▼"/>	
<input type="button" value="1"/> <input type="button" value="2"/> <input type="button" value="3"/> <input type="button" value="4"/> ... <input type="button" value="10"/> <input type="button" value="11"/> <input type="button" value=">"/>	

Customizing the stacked ring width

The default ring width is `3`, but you may customize it to be from `0` to `8`:

```
ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->ring(5)
```

Customizing the stacked overlap

The default overlap is `4`, but you may customize it to be from `0` to `8`:

```
ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->overlap(2)
```

Wrapping multiple images

Images can be set to wrap if they can't fit on one line, by setting `wrap()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->wrap()
```

Note: the "width" for wrapping is affected by the column label, so you may need to use a shorter or hidden label to wrap more tightly.

Setting a limit

You may limit the maximum number of images you want to display by passing `limit()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->limit(3)
```

Name	Colleagues
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	
Showing 1 to 5 of 51 results <div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 10 11 > </div>	

Showing the remaining images count

When you set a limit you may also display the count of remaining images by passing `limitedRemainingText()`.

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText()
```

Name	Colleagues
Dan Harrin	
Ryan Chandler	
Zep Fietje	
Dennis Koch	
Adam Weston	
Showing 1 to 5 of 51 results <div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 10 11 > </div>	

Showing the limited remaining text separately

By default, `limitedRemainingText()` will display the count of remaining images as a number stacked on the other images. If you prefer to show the count as a number after the images, you may use the `isSeparate: true` parameter:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText(isSeparate: true)
```

Name	Colleagues
Dan Harrin	 +2
Ryan Chandler	 +2
Zep Fietje	 +1
Dennis Koch	 +1
Adam Weston	 +1

Showing 1 to 5 of 51 results Per page 5 ▾ 1 2 3 4 ... 10 11 >

Customizing the limited remaining text size

By default, the size of the remaining text is `sm`. You can customize this to be `xs`, `md` or `lg` using the `size` parameter:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('colleagues.avatar')
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText(size: 'lg')
```

Custom attributes

You may customize the extra HTML attributes of the image using `extraImgAttributes()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('logo')
    ->extraImgAttributes(['loading' => 'lazy']),
```

You can access the current record using a `$record` parameter:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('logo')
->extraImgAttributes(fn (Company $record): array => [
    'alt' => "{$record->name} logo",
]) ,
```

Prevent file existence checks

When the table is loaded, it will automatically detect whether the images exist. This is all done on the backend. When using remote storage with many images, this can be time-consuming. You can use the `checkFileExistence(false)` method to disable this feature:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('attachment')
->checkFileExistence(false)
```

Color

Overview

The color column allows you to show the color preview from a CSS color definition, typically entered using the color picker field, in one of the supported formats (HEX, HSL, RGB, RGBA).

```
use Filament\Tables\Columns\ColorColumn;

ColorColumn::make('color')
```

Title	Color
What is Filament?	
Top 5 best features of Filament	
Tips for building a great Filament plugin	
Customizing Filament's UI with a theme	
New Filament plugins in August	

Showing 1 to 5 of 50 results

Per page 5 ▾
1
2
3
4
...
9
10
>

Allowing the color to be copied to the clipboard

You may make the color copyable, such that clicking on the preview copies the CSS value to the clipboard, and optionally specify a custom confirmation message and duration in milliseconds. This feature only works when SSL is enabled for the app.

```
use Filament\Tables\Columns\ColorColumn;

ColorColumn::make('color')
    ->copyable()
    ->copyMessage('Color code copied')
    ->copyMessageDuration(1500)
```

Title	Color
What is Filament?	
Top 5 best features of Filament	
Tips for building a great Filament plugin	
Customizing Filament's UI with a theme	
New Filament plugins in August	

Showing 1 to 5 of 50 results Per page: 5 1 2 3 4 ... 9 10 >

Customizing the text that is copied to the clipboard

You can customize the text that gets copied to the clipboard using the `copyableState()` method:

```
use Filament\Tables\Columns\ColorColumn;

ColorColumn::make('color')
    ->copyable()
    ->copyableState(fn (string $state): string => "Color: {$state}")
```

In this function, you can access the whole table row with `$record`:

```
use App\Models\Post;
use Filament\Tables\Columns\ColorColumn;

ColorColumn::make('color')
    ->copyable()
    ->copyableState(fn (Post $record): string => "Color: {$record->color}")
```

Wrapping multiple color blocks

Color blocks can be set to wrap if they can't fit on one line, by setting `wrap()`:

```
use Filament\Tables\Columns\ColorColumn;

ColorColumn::make('color')
    ->wrap()
```

Note: the "width" for wrapping is affected by the column label, so you may need to use a shorter or hidden label to wrap more tightly.

Select

Overview

The select column allows you to render a select field inside the table, which can be used to update that database record without needing to open a new page or a modal.

You must pass options to the column:

```
use Filament\Tables\Columns\SelectColumn;

SelectColumn::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
)
```

Title	Status
What is Filament?	Published ▾
Top 5 best features of Filament	Reviewing ▾
Tips for building a great Filament plugin	Draft ▾
Customizing Filament's UI with a theme	Reviewing ▾
New Filament plugins in August	Published ▾
Showing 1 to 5 of 50 results	<div style="display: flex; justify-content: space-between;"> Per page 5 ▾ 1 2 3 4 ... 9 10 > </div>

Validation

You can validate the input by passing any [Laravel validation rules](#) in an array:

```
use Filament\Tables\Columns\SelectColumn;

SelectColumn::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->rules(['required'])
)
```

Disabling placeholder selection

You can prevent the placeholder from being selected using the `selectablePlaceholder()` method:

```
use Filament\Tables\Columns>SelectColumn;

SelectColumn::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->selectablePlaceholder(false)
```

Lifecycle hooks

Hooks may be used to execute code at various points within the select's lifecycle:

```
SelectColumn::make()
    ->beforeStateUpdated(function ($record, $state) {
        // Runs before the state is saved to the database.
    })
    ->afterStateUpdated(function ($record, $state) {
        // Runs after the state is saved to the database.
    })
```

Toggle

Overview

The toggle column allows you to render a toggle button inside the table, which can be used to update that database record without needing to open a new page or a modal:

```
use Filament\Tables\Columns\ToggleColumn;

ToggleColumn::make('is_admin')
```

Name	Is admin
Dan Harrin	<input checked="" type="checkbox"/>
Ryan Chandler	<input checked="" type="checkbox"/>
Zep Fietje	<input type="checkbox"/>
Dennis Koch	<input type="checkbox"/>
Adam Weston	<input type="checkbox"/>

Showing 1 to 5 of 51 results Per page: 5 [1](#) [2](#) [3](#) [4](#) ... [10](#) [11](#) >

Lifecycle hooks

Hooks may be used to execute code at various points within the toggle's lifecycle:

```
ToggleColumn::make()
    ->beforeStateUpdated(function ($record, $state) {
        // Runs before the state is saved to the database.
    })
    ->afterStateUpdated(function ($record, $state) {
        // Runs after the state is saved to the database.
    })
```

Text Input

Overview

The text input column allows you to render a text input inside the table, which can be used to update that database record without needing to open a new page or a modal:

```
use Filament\Tables\Columns\TextInputColumn;

TextInputColumn::make('email')
```

Name	Email
Dan Harrin	dan@filamentphp.com
Ryan Chandler	ryan@filamentphp.com
Zep Fietje	zep@filamentphp.com
Dennis Koch	dennis@filamentphp.co
Adam Weston	adam@filamentphp.com
Showing 1 to 5 of 51 results Per page 5 1 2 3 4 ... 10 11 > 	

Validation

You can validate the input by passing any [Laravel validation rules](#) in an array:

```
use Filament\Tables\Columns\TextInputColumn;

TextInputColumn::make('name')
    ->rules(['required', 'max:255'])
```

Customizing the HTML input type

You may use the `type()` method to pass a custom [HTML input type](#):

```
use Filament\Tables\Columns\TextInputColumn;

TextInputColumn::make('background_color')->type('color')
```

Lifecycle hooks

Hooks may be used to execute code at various points within the input's lifecycle:

```
TextInputColumn::make()
->beforeStateUpdated(function ($record, $state) {
    // Runs before the state is saved to the database.
})
->afterStateUpdated(function ($record, $state) {
    // Runs after the state is saved to the database.
})
```

Checkbox

Overview

The checkbox column allows you to render a checkbox inside the table, which can be used to update that database record without needing to open a new page or a modal:

```
use Filament\Tables\Columns\CheckboxColumn;

CheckboxColumn::make('is_admin')
```

Name	Is admin
Dan Harrin	<input checked="" type="checkbox"/>
Ryan Chandler	<input checked="" type="checkbox"/>
Zep Fietje	<input type="checkbox"/>
Dennis Koch	<input type="checkbox"/>
Adam Weston	<input type="checkbox"/>

Showing 1 to 5 of 51 results

Per page 5 ▾

1 2 3 4 ... 10 11 >

Lifecycle hooks

Hooks may be used to execute code at various points within the checkbox's lifecycle:

```
CheckboxColumn::make()
->beforeStateUpdated(function ($record, $state) {
    // Runs before the state is saved to the database.
})
->afterStateUpdated(function ($record, $state) {
    // Runs after the state is saved to the database.
})
```

Custom

View columns

You may render a custom view for a cell using the `view()` method:

```
use Filament\Tables\Columns\ViewColumn;

ViewColumn::make('status')->view('filament.tables.columns.status-switcher')
```

This assumes that you have a `resources/views/filament/tables/columns/status-switcher.blade.php` file.

Custom classes

You may create your own custom column classes and cell views, which you can reuse across your project, and even release as a plugin to the community.

If you're just creating a simple custom column to use once, you could instead use a [view column](#) to render any custom Blade file.

To create a custom column class and view, you may use the following command:

```
php artisan make:table-column StatusSwitcher
```

This will create the following column class:

```
use Filament\Tables\Columns\Column;

class StatusSwitcher extends Column
{
    protected string $view = 'filament.tables.columns.status-switcher';
}
```

It will also create a view file at `resources/views/filament/tables/columns/status-switcher.blade.php`.

Accessing the state

Inside your view, you may retrieve the state of the cell using the `$getState()` function:

```
<div>
    {{ $getState() }}
</div>
```

Accessing the Eloquent record

Inside your view, you may access the Eloquent record using the `$getRecord()` function:

```
<div>
{{ $getRecord() ->name }}</div>
```

Relationships

Displaying data from relationships

You may use "dot notation" to access columns within relationships. The name of the relationship comes first, followed by a period, followed by the name of the column to display:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('author.name')
```

Counting relationships

If you wish to count the number of related records in a column, you may use the `counts()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('users_count')->counts('users')
```

In this example, `users` is the name of the relationship to count from. The name of the column must be `users_count`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('users_count')->counts([
    'users' => fn (Builder $query) => $query->where('is_active', true),
])
```

Determining relationship existence

If you simply wish to indicate whether related records exist in a column, you may use the `exists()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('users_exists')->exists('users')
```

In this example, `users` is the name of the relationship to check for existence. The name of the column must be `users_exists`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('users_exists')->exists([
    'users' => fn (Builder $query) => $query->where('is_active', true),
])
])
```

Aggregating relationships

Filament provides several methods for aggregating a relationship field, including `avg()`, `max()`, `min()` and `sum()`. For instance, if you wish to show the average of a field on all related records in a column, you may use the `avg()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('users_avg_age')->avg('users', 'age')
```

In this example, `users` is the name of the relationship, while `age` is the field that is being averaged. The name of the column must be `users_avg_age`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('users_avg_age')->avg([
    'users' => fn (Builder $query) => $query->where('is_active', true),
], 'age')
```

Advanced

Table column utility injection

The vast majority of methods used to configure columns accept functions as parameters instead of hardcoded values:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')
    ->color(fn (string $state): string => match ($state) {
        'draft' => 'gray',
        'reviewing' => 'warning',
        'published' => 'success',
        'rejected' => 'danger',
    })
}
```

This alone unlocks many customization possibilities.

The package is also able to inject many utilities to use inside these functions, as parameters. All customization methods that accept functions as arguments can inject utilities.

These injected utilities require specific parameter names to be used. Otherwise, Filament doesn't know what to inject.

Injecting the current state of a column

If you wish to access the current state (value) of the column, define a `[$state]` parameter:

```
function ($state) {
    // ...
}
```

Injecting the current Eloquent record

If you wish to access the current Eloquent record of the column, define a `[$record]` parameter:

```
use Illuminate\Database\Eloquent\Model;

function (Model $record) {
    // ...
}
```

Be aware that this parameter will be `null` if the column is not bound to an Eloquent record. For instance, the `label()` method of a column will not have access to the record, as the label is not related to any table row.

Injecting the current column instance

If you wish to access the current column instance, define a `[$column]` parameter:

```
use Filament\Tables\Columns\Column;

function (Column $column) {
    // ...
}
```

Injecting the current Livewire component instance

If you wish to access the current Livewire component instance that the table belongs to, define a `$livewire` parameter:

```
use Filament\Tables\Contracts\HasTable;

function (HasTable $livewire) {
    // ...
}
```

Injecting the current table instance

If you wish to access the current table configuration instance that the column belongs to, define a `$table` parameter:

```
use Filament\Tables\Table;

function (Table $table) {
    // ...
}
```

Injecting the current table row loop

If you wish to access the current [Laravel Blade loop object](#) that the column is rendered part of, define a `$rowLoop` parameter:

```
function (stdClass $rowLoop) {
    // ...
}
```

As `$rowLoop` is Laravel Blade's `$loop` object, you can access the current row index using `$rowLoop->index`. Similar to `$record`, this parameter will be `null` if the column is currently being rendered outside a table row.

Injecting multiple utilities

The parameters are injected dynamically using reflection, so you are able to combine multiple parameters in any order:

```
use Filament\Tables\Contracts\HasTable;
use Illuminate\Database\Eloquent\Model;

function (HasTable $livewire, Model $record) {
    // ...
}
```

Injecting dependencies from Laravel's container

You may inject anything from Laravel's container like normal, alongside utilities:

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Http\Request;

function (Request $request, Model $record) {
    // ...
}
```

Filters

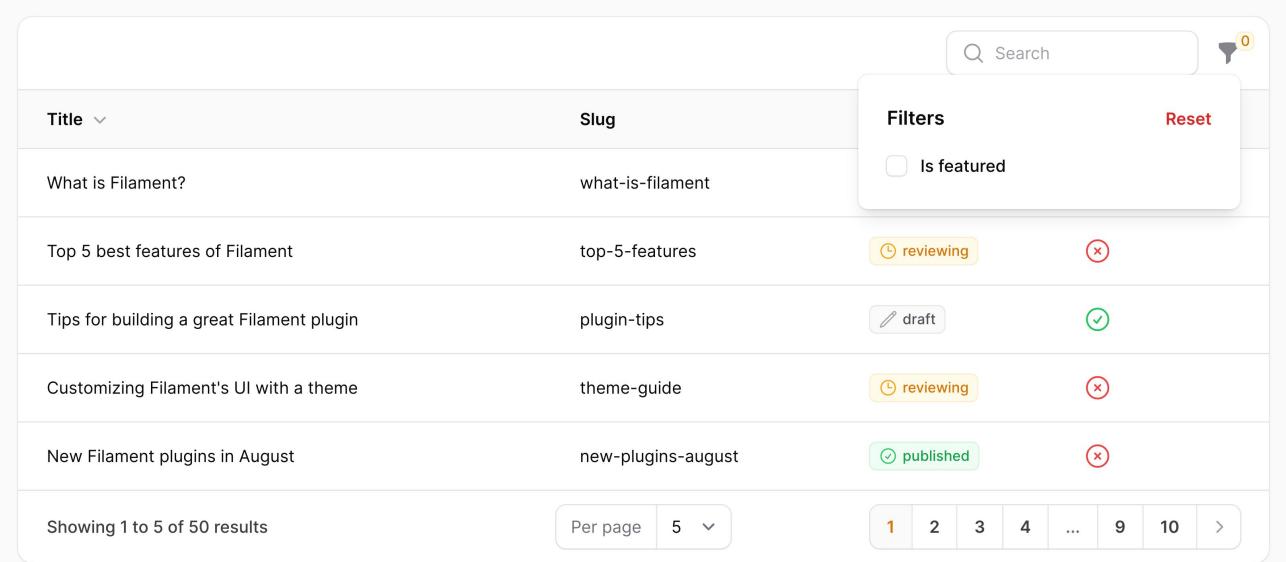
Getting Started

Overview

Filters allow you to define certain constraints on your data, and allow users to scope it to find the information they need. You put them in the `$table->filters()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ]);
}
```



The screenshot shows a table component from the Filament UI library. The table has two columns: 'Title' and 'Slug'. Below the table, there's a search bar and a filter section. The filter section contains a checkbox labeled 'Is featured' and a 'Reset' button. There are also buttons for 'reviewing' and 'published' status, and a 'draft' button with a checkmark. At the bottom, there's a pagination control showing 'Showing 1 to 5 of 50 results' and a page navigation bar with buttons for 1, 2, 3, 4, ..., 9, 10, >.

Title	Slug	Filters	Reset
What is Filament?	what-is-filament	<input type="checkbox"/> Is featured	
Top 5 best features of Filament	top-5-features	⌚ reviewing	×
Tips for building a great Filament plugin	plugin-tips	✍ draft	✓
Customizing Filament's UI with a theme	theme-guide	⌚ reviewing	×
New Filament plugins in August	new-plugins-august	⌚ published	×

Filters may be created using the static `make()` method, passing its unique name. You should then pass a callback to `query()` which applies your filter's scope:

```
use Filament\Tables\Filters\Filter;
use Illuminate\Database\Eloquent\Builder;

Filter::make('is_featured')
    ->query(fn (Builder $query): Builder => $query->where('is_featured', true))
```

Available filters

By default, using the `Filter::make()` method will render a checkbox form component. When the checkbox is on, the `query()` will be activated.

- You can also [replace the checkbox with a toggle](#).
- You can use a [ternary filter](#) to replace the checkbox with a select field to allow users to pick between 3 states - usually "true", "false" and "blank". This is useful for filtering boolean columns that are nullable.
- The [trashed filter](#) is a pre-built ternary filter that allows you to filter soft-deletable records.
- You may use a [select filter](#) to allow users to select from a list of options, and filter using the selection.
- You may use a [query builder](#) to allow users to create complex sets of filters, with an advanced user interface for combining constraints.
- You may build [custom filters](#) with other form fields, to do whatever you want.

Setting a label

By default, the label of the filter, which is displayed in the filter form, is generated from the name of the filter. You may customize this using the `label()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')
->label('Featured')
```

Optionally, you can have the label automatically translated [using Laravel's localization features](#) with the `translateLabel()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')
->translateLabel() // Equivalent to `label(__('Is featured'))`
```

Customizing the filter form

By default, creating a filter with the `Filter` class will render a [checkbox form component](#). When the checkbox is checked, the `query()` function will be applied to the table's query, scoping the records in the table. When the checkbox is unchecked, the `query()` function will be removed from the table's query.

Filters are built entirely on Filament's form fields. They can render any combination of form fields, which users can then interact with to filter the table.

Using a toggle button instead of a checkbox

The simplest example of managing the form field that is used for a filter is to replace the [checkbox](#) with a [toggle button](#), using the `toggle()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')
->toggle()
```

The screenshot shows a Filament 3.x dashboard interface. At the top right is a search bar with a magnifying glass icon and the word "Search". Next to it is a trash can icon with a "0" notification. Below the search bar is a "Filters" section with a "Reset" button. A toggle switch labeled "Is featured" is turned on. The main area displays a table of posts with columns "Title" and "Slug". Each post has a status badge and a delete button. The posts listed are:

Title	Slug	Status	Action
What is Filament?	what-is-filament		X
Top 5 best features of Filament	top-5-features	🕒 reviewing	X
Tips for building a great Filament plugin	plugin-tips	✍ draft	✓
Customizing Filament's UI with a theme	theme-guide	🕒 reviewing	X
New Filament plugins in August	new-plugins-august	🕒 published	X

At the bottom left, it says "Showing 1 to 5 of 50 results". To the right is a "Per page" dropdown set to "5" and a navigation bar with pages 1 through 10.

Applying the filter by default

You may set a filter to be enabled by default, using the `default()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')
->default()
```

Customizing the built-in filter form field

Whether you are using a checkbox, a `toggle` or a `select`, you can customize the built-in form field used for the filter, using the `modifyFormFieldUsing()` method. The method accepts a function with a `$field` parameter that gives you access to the form field object to customize:

```
use Filament\Forms\Components\Checkbox;
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')
->modifyFormFieldUsing(fn (Checkbox $field) => $field->inline(false))
```

Persist filters in session

To persist the table filters in the user's session, use the `persistFiltersInSession()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->persistFiltersInSession();
}
```

Deferring filters

You can defer filter changes from affecting the table, until the user clicks an "Apply" button. To do this, use the `deferFilters()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->deferFilters();
}
```

Customizing the apply filters action

When deferring filters, you can customize the "Apply" button, using the `filtersApplyAction()` method, passing a closure that returns an action. All methods that are available to [customize action trigger buttons](#) can be used:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->filtersApplyAction(
            fn (Action $action) => $action
                ->link()
                ->label('Save filters to table'),
        );
}
```

Deselecting records when filters change

By default, all records will be deselected when the filters change. Using the `deselectAllRecordsWhenFiltered(false)` method, you can disable this behavior:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->deselectAllRecordsWhenFiltered(false);
}
```

Modifying the base query

By default, modifications to the Eloquent query performed in the `query()` method will be applied inside a scoped `where()` clause. This is to ensure that the query does not clash with any other filters that may be applied, especially those that use `orWhere()`.

However, the downside of this is that the `query()` method cannot be used to modify the query in other ways, such as removing global scopes, since the base query needs to be modified directly, not the scoped query.

To modify the base query directly, you may use the `baseQuery()` method, passing a closure that receives the base query:

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\SoftDeletingScope;
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('trashed')
    // ...
    ->baseQuery(fn (Builder $query) => $query->withoutGlobalScopes([
        SoftDeletingScope::class,
    ]))
```

Customizing the filters trigger action

To customize the filters trigger buttons, you may use the `filtersTriggerAction()` method, passing a closure that returns an action. All methods that are available to [customize action trigger buttons](#) can be used:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->filtersTriggerAction(
            fn (Action $action) => $action
                ->button()
                ->label('Filter'),
        );
}
```

Title	Slug	Status	Is featured
What is Filament?	what-is-filament	published	✓
Top 5 best features of Filament	top-5-features	reviewing	✗
Tips for building a great Filament plugin	plugin-tips	draft	✓
Customizing Filament's UI with a theme	theme-guide	reviewing	✗
New Filament plugins in August	new-plugins-august	published	✗

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Table filter utility injection

The vast majority of methods used to configure filters accept functions as parameters instead of hardcoded values:

```
use App\Models\Author;
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('author')
    ->options(fn (): array => Author::query()->pluck('name', 'id')->all())
```

This alone unlocks many customization possibilities.

The package is also able to inject many utilities to use inside these functions, as parameters. All customization methods that accept functions as arguments can inject utilities.

These injected utilities require specific parameter names to be used. Otherwise, Filament doesn't know what to inject.

Injecting the current filter instance

If you wish to access the current filter instance, define a `$filter` parameter:

```
use Filament\Tables\Filters\BaseFilter;

function (BaseFilter $filter) {
    // ...
}
```

Injecting the current Livewire component instance

If you wish to access the current Livewire component instance that the table belongs to, define a `$livewire` parameter:

```
use Filament\Tables\Contracts\HasTable;

function (HasTable $livewire) {
    // ...
}
```

Injecting the current table instance

If you wish to access the current table configuration instance that the filter belongs to, define a `$table` parameter:

```
use Filament\Tables\Table;

function (Table $table) {
    // ...
}
```

Injecting multiple utilities

The parameters are injected dynamically using reflection, so you are able to combine multiple parameters in any order:

```
use Filament\Tables\Contracts\HasTable;
use Filament\Tables\Table;

function (HasTable $livewire, Table $table) {
    // ...
}
```

Injecting dependencies from Laravel's container

You may inject anything from Laravel's container like normal, alongside utilities:

```
use Filament\Tables\Table;
use Illuminate\Http\Request;

function (Request $request, Table $table) {
    // ...
}
```

Select

Overview

Often, you will want to use a [select field](#) instead of a checkbox. This is especially true when you want to filter a column based on a set of pre-defined options that the user can choose from. To do this, you can create a filter using the `SelectFilter` class:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
```

The `options()` that are passed to the filter are the same as those that are passed to the [select field](#).

Customizing the column used by a select filter

Select filters do not require a custom `query()` method. The column name used to scope the query is the name of the filter. To customize this, you may use the `attribute()` method:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->attribute('status_id')
```

Multi-select filters

These allow the user to select multiple options to apply the filter to their table. For example, a status filter may present the user with a few status options to pick from and filter the table using. When the user selects multiple options, the table will be filtered to show records that match any of the selected options. You can enable this behavior using the `multiple()` method:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('status')
    ->multiple()
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
])
```

Relationship select filters

Select filters are also able to automatically populate themselves based on a relationship. For example, if your table has a `author` relationship with a `name` column, you may use `relationship()` to filter the records belonging to an author:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('author')
    ->relationship('author', 'name')
```

Preloading the select filter relationship options

If you'd like to populate the searchable options from the database when the page is loaded, instead of when the user searches, you can use the `preload()` method:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('author')
    ->relationship('author', 'name')
    ->searchable()
    ->preload()
```

Customizing the select filter relationship query

You may customize the database query that retrieves options using the third parameter of the `relationship()` method:

```
use Filament\Tables\Filters\SelectFilter;
use Illuminate\Database\Eloquent\Builder;

SelectFilter::make('author')
    ->relationship('author', 'name', fn (Builder $query) => $query->withTrashed())
```

Searching select filter options

You may enable a search input to allow easier access to many options, using the `searchable()` method:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('author')
    ->relationship('author', 'name')
    ->searchable()
```

Disable placeholder selection

You can remove the placeholder (null option), which disables the filter so all options are applied, using the `selectablePlaceholder()` method:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->default('draft')
    ->selectablePlaceholder(false)
```

Ternary

Overview

Ternary filters allow you to easily create a select filter which has three states - usually true, false and blank. To filter a column named `is_admin` to be `true` or `false`, you may use the ternary filter:

```
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('is_admin')
```

Using a ternary filter with a nullable column

Another common pattern is to use a nullable column. For example, when filtering verified and unverified users using the `email_verified_at` column, unverified users have a null timestamp in this column. To apply that logic, you may use the `nullable()` method:

```
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('email_verified_at')
    ->nullable()
```

Customizing the column used by a ternary filter

The column name used to scope the query is the name of the filter. To customize this, you may use the `attribute()` method:

```
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('verified')
    ->nullable()
    ->attribute('status_id')
```

Customizing the ternary filter option labels

You may customize the labels used for each state of the ternary filter. The true option label can be customized using the `trueLabel()` method. The false option label can be customized using the `falseLabel()` method. The blank (default) option label can be customized using the `placeholder()` method:

```
use Illuminate\Database\Eloquent\Builder;
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('email_verified_at')
    ->label('Email verification')
    ->nullable()
    ->placeholder('All users')
    ->trueLabel('Verified users')
    ->falseLabel('Not verified users')
```

Customizing how a ternary filter modifies the query

You may customize how the query changes for each state of the ternary filter, use the `queries()` method:

```
use Illuminate\Database\Eloquent\Builder;
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('email_verified_at')
    ->label('Email verification')
    ->placeholder('All users')
    ->trueLabel('Verified users')
    ->falseLabel('Not verified users')
    ->queries(
        true: fn (Builder $query) => $query->whereNotNull('email_verified_at'),
        false: fn (Builder $query) => $query->whereNull('email_verified_at'),
        blank: fn (Builder $query) => $query, // In this example, we do not want to filter the
query when it is blank.
    )
```

Filtering soft deletable records

The `TrashedFilter` can be used to filter soft deleted records. It is a type of ternary filter that is built-in to Filament. You can use it like so:

```
use Filament\Tables\Filters\TrashedFilter;

TrashedFilter::make()
```

QueryBuilder

Overview

The query builder allows you to define a complex set of conditions to filter the data in your table. It is able to handle unlimited nesting of conditions, which you can group together with "and" and "or" operations.

To use it, you need to define a set of "constraints" that will be used to filter the data. Filament includes some built-in constraints, that follow common data types, but you can also define your own custom constraints.

You can add a query builder to any table using the `QueryBuilder` filter:

```
use Filament\Tables\Filters\QueryBuilder;
use Filament\Tables\Filters\QueryBuilder\Constraints\BooleanConstraint;
use Filament\Tables\Filters\QueryBuilder\Constraints\DateConstraint;
use Filament\Tables\Filters\QueryBuilder\Constraints\NumberConstraint;
use Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint;
use
Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint\Operators\IsRelatedToOpera

use Filament\Tables\Filters\QueryBuilder\Constraints\SelectConstraint;
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

QueryBuilder::make()
->constraints([
    TextConstraint::make('name'),
    BooleanConstraint::make('is_visible'),
    NumberConstraint::make('stock'),
    SelectConstraint::make('status')
        ->options([
            'draft' => 'Draft',
            'reviewing' => 'Reviewing',
            'published' => 'Published',
        ])
        ->multiple(),
    DateConstraint::make('created_at'),
    RelationshipConstraint::make('categories')
        ->multiple()
        ->selectable(
            IsRelatedToOperator::make()
                ->titleAttribute('name')
                ->searchable()
                ->multiple(),
        ),
    NumberConstraint::make('reviewsRating')
        ->relationship('reviews', 'rating')
        ->integer(),
])
])
```

When deeply nesting the query builder, you might need to increase the amount of space that the filters can consume. One way of doing this is to [position the filters above the table content](#):

```

use Filament\Tables\Enums\FiltersLayout;
use Filament\Tables\Filters\QueryBuilder;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            QueryBuilder::make()
                ->constraints([
                    // ...
                ]),
            layout: FiltersLayout::AboveContent);
}

```

Available constraints

Filament ships with many different constraints that you can use out of the box. You can also [create your own custom constraints](#):

- [Text constraint](#)
- [Boolean constraint](#)
- [Number constraint](#)
- [Date constraint](#)
- [Select constraint](#)
- [Relationship constraint](#)

Text constraints

Text constraints allow you to filter text fields. They can be used to filter any text field, including via relationships.

```

use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('name') // Filter the `name` column

TextConstraint::make('creatorName')
    ->relationship(name: 'creator', titleAttribute: 'name') // Filter the `name` column on the
`creator` relationship

```

By default, the following operators are available:

- Contains - filters a column to contain the search term
- Does not contain - filters a column to not contain the search term
- Starts with - filters a column to start with the search term
- Does not start with - filters a column to not start with the search term
- Ends with - filters a column to end with the search term
- Does not end with - filters a column to not end with the search term
- Equals - filters a column to equal the search term
- Does not equal - filters a column to not equal the search term
- Is filled - filters a column to not be empty
- Is blank - filters a column to be empty

Boolean constraints

Boolean constraints allow you to filter boolean fields. They can be used to filter any boolean field, including via relationships.

```
use Filament\Tables\Filters\QueryBuilder\Constraints\BooleanConstraint;

BooleanConstraint::make('is_visible') // Filter the `is_visible` column

BooleanConstraint::make('creatorIsAdmin')
    ->relationship(name: 'creator', titleAttribute: 'is_admin') // Filter the `is_admin` column
on the `creator` relationship
```

By default, the following operators are available:

- Is true - filters a column to be `true`
- Is false - filters a column to be `false`

Number constraints

Number constraints allow you to filter numeric fields. They can be used to filter any numeric field, including via relationships.

```
use Filament\Tables\Filters\QueryBuilder\Constraints\NumberConstraint;

NumberConstraint::make('stock') // Filter the `stock` column

NumberConstraint::make('ordersItemCount')
    ->relationship(name: 'orders', titleAttribute: 'item_count') // Filter the `item_count` column
on the `orders` relationship
```

By default, the following operators are available:

- Is minimum - filters a column to be greater than or equal to the search number
- Is less than - filters a column to be less than the search number
- Is maximum - filters a column to be less than or equal to the search number
- Is greater than - filters a column to be greater than the search number
- Equals - filters a column to equal the search number
- Does not equal - filters a column to not equal the search number
- Is filled - filters a column to not be empty
- Is blank - filters a column to be empty

When using `relationship()` with a number constraint, users also have the ability to "aggregate" related records. This means that they can filter the column to be the sum, average, minimum or maximum of all the related records at once.

Integer constraints

By default, number constraints will allow decimal values. If you'd like to only allow integer values, you can use the `integer()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\NumberConstraint;

NumberConstraint::make('stock')
    ->integer()
```

Date constraints

Date constraints allow you to filter date fields. They can be used to filter any date field, including via relationships.

```
use Filament\Tables\Filters\QueryBuilder\Constraints\DateConstraint;

DateConstraint::make('created_at') // Filter the `created_at` column

DateConstraint::make('creatorCreatedAt')
    ->relationship(name: 'creator', titleAttribute: 'created_at') // Filter the `created_at` column on the `creator` relationship
```

By default, the following operators are available:

- Is after - filters a column to be after the search date
- Is not after - filters a column to not be after the search date, or to be the same date
- Is before - filters a column to be before the search date
- Is not before - filters a column to not be before the search date, or to be the same date
- Is date - filters a column to be the same date as the search date
- Is not date - filters a column to not be the same date as the search date
- Is month - filters a column to be in the same month as the selected month
- Is not month - filters a column to not be in the same month as the selected month
- Is year - filters a column to be in the same year as the searched year
- Is not year - filters a column to not be in the same year as the searched year

Select constraints

Select constraints allow you to filter fields using a select field. They can be used to filter any field, including via relationships.

```
use Filament\Tables\Filters\QueryBuilder\Constraints\SelectConstraint;

SelectConstraint::make('status') // Filter the `status` column
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])

SelectConstraint::make('creatorStatus')
    ->relationship(name: 'creator', titleAttribute: 'department') // Filter the `department` column on the `creator` relationship
    ->options([
        'sales' => 'Sales',
        'marketing' => 'Marketing',
        'engineering' => 'Engineering',
        'purchasing' => 'Purchasing',
    ])
```

Searchable select constraints

By default, select constraints will not allow the user to search the options. If you'd like to allow the user to search the options, you can use the `searchable()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\SelectConstraint;

SelectConstraint::make('status')
    ->searchable()
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
])
```

Multi-select constraints

By default, select constraints will only allow the user to select a single option. If you'd like to allow the user to select multiple options, you can use the `multiple()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\SelectConstraint;

SelectConstraint::make('status')
    ->multiple()
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
])
```

When the user selects multiple options, the table will be filtered to show records that match any of the selected options.

Relationship constraints

Relationship constraints allow you to filter fields using data about a relationship:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint;
use
Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint\Operators\IsRelatedToOpera

RelationshipConstraint::make('creator') // Filter the `creator` relationship
->selectable(
    IsRelatedToOperator::make()
        ->titleAttribute('name')
        ->searchable()
        ->multiple(),
)
```

The `IsRelatedToOperator` is used to configure the "Is / Contains" and "Is not / Does not contain" operators. It provides a select field which allows the user to filter the relationship by which records are attached to it. The `titleAttribute()` method is used to specify which attribute should be used to identify each related record in the list. The `searchable()` method makes the list searchable. The `multiple()` method allows the user to select multiple related records, and if they do, the table will be filtered to show records that match any of the selected related records.

Multiple relationships

By default, relationship constraints only include operators that are appropriate for filtering a singular relationship, like a `BelongsTo`. If you have a relationship such as a `HasMany` or `BelongsToMany`, you may wish to mark the constraint as `multiple()`:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint;

RelationshipConstraint::make('categories')
->multiple()
```

This will add the following operators to the constraint:

- Has minimum - filters a column to have at least the specified number of related records
- Has less than - filters a column to have less than the specified number of related records
- Has maximum - filters a column to have at most the specified number of related records
- Has more than - filters a column to have more than the specified number of related records
- Has - filters a column to have the specified number of related records
- Does not have - filters a column to not have the specified number of related records

Empty relationship constraints

The `RelationshipConstraint` does not support `nullable()` in the same way as other constraints.

If the relationship is `multiple()`, then the constraint will show an option to filter out "empty" relationships. This means that the relationship has no related records. If your relationship is singular, then you can use the `emptyable()` method to show an option to filter out "empty" relationships:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint;

RelationshipConstraint::make('creator')
->emptyable()
```

If you have a `multiple()` relationship that must always have at least 1 related record, then you can use the `emptyable(false)` method to hide the option to filter out "empty" relationships:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\RelationshipConstraint;

RelationshipConstraint::make('categories')
->emptyable(false)
```

Nullable constraints

By default, constraints will not show an option to filter `null` values. If you'd like to show an option to filter `null` values, you can use the `nullable()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('name')
->nullable()
```

Now, the following operators are also available:

- Is filled - filters a column to not be empty
- Is blank - filters a column to be empty

Scoping relationships

When you use the `relationship()` method on a constraint, you can scope the relationship to filter the related records using the `modifyQueryUsing` argument:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;
use Illuminate\Database\Eloquent\Builder;

TextConstraint::make('adminCreatorName')
    ->relationship(
        name: 'creator',
        titleAttribute: 'name',
        modifyQueryUsing: fn (Builder $query) => $query->where('is_admin', true),
    )
)
```

Customizing the constraint icon

Each constraint type has a default `icon`, which is displayed next to the label in the picker. You can customize the icon for a constraint by passing its name to the `icon()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('author')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->icon('heroicon-m-user')
```

Overriding the default operators

Each constraint type has a set of default operators, which you can customize by using the `operators()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Operators\IsFilledOperator;
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('author')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->operators([
        IsFilledOperator::make(),
    ])
```

This will remove all operators, and register the `EqualsOperator`.

If you'd like to add an operator to the end of the list, use `pushOperators()` instead:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Operators\IsFilledOperator;
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('author')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->pushOperators([
        IsFilledOperator::class,
    ])
```

If you'd like to add an operator to the start of the list, use `unshiftOperators()` instead:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Operators\IsFilledOperator;
use Filament\Tables\Filters\QueryBuilder\Constraints\TextConstraint;

TextConstraint::make('author')
    ->relationship(name: 'author', titleAttribute: 'name')
    ->unshiftOperators([
        IsFilledOperator::class,
    ])
)
```

Creating custom constraints

Custom constraints can be created "inline" with other constraints using the `Constraint::make()` method. You should also pass an `icon` to the `icon()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Constraint;

Constraint::make('subscribed')
    ->icon('heroicon-m-bell')
    ->operators([
        // ...
    ]),
```

If you want to customize the label of the constraint, you can use the `label()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Constraint;

Constraint::make('subscribed')
    ->label('Subscribed to updates')
    ->icon('heroicon-m-bell')
    ->operators([
        // ...
    ]),
```

Now, you must define operators for the constraint. These are options that you can pick from to filter the column. If the column is nullable, you can also register that built-in operator for your custom constraint:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Constraint;
use Filament\Tables\Filters\QueryBuilder\Constraints\Operators\IsFilledOperator;

Constraint::make('subscribed')
    ->label('Subscribed to updates')
    ->icon('heroicon-m-bell')
    ->operators([
        // ...
        IsFilledOperator::class,
    ]),
```

Creating custom operators

Custom operators can be created using the `Operator::make()` method:

```
use Filament\Tables\Filters\QueryBuilder\Constraints\Operators\Operator;

Operator::make('subscribed')
    ->label(fn (bool $isInverse): string => $isInverse ? 'Not subscribed' : 'Subscribed')
    ->summary(fn (bool $isInverse): string => $isInverse ? 'You are not subscribed' : 'You are
subscribed')
    ->baseQuery(fn (Builder $query, bool $isInverse) => $query->[$isInverse ? 'whereDoesntHave'
: 'whereHas'](
        'subscriptions.user',
        fn (Builder $query) => $query->whereKey(auth()->user()),
    )),
)
```

In this example, the operator is able to filter records based on whether or not the authenticated user is subscribed to the record. A subscription is recorded in the `subscriptions` relationship of the table.

The `baseQuery()` method is used to define the query that will be used to filter the records. The `$isInverse` argument is `false` when the "Subscribed" option is selected, and `true` when the "Not subscribed" option is selected. The function is applied to the base query of the table, where `whereHas()` can be used. If your function does not need to be applied to the base query of the table, like when you are using a simple `where()` or `whereIn()`, you can use the `query()` method instead, which has the bonus of being able to be used inside nested "OR" groups.

The `label()` method is used to render the options in the operator select. Two options are registered for each operator, one for when the operator is not inverted, and one for when it is inverted.

The `summary()` method is used in the header of the constraint when it is applied to the query, to provide an overview of the active constraint.

Customizing the constraint picker

Changing the number of columns in the constraint picker

The constraint picker has only 1 column. You may customize it by passing a number of columns to

`constraintPickerColumns()`:

```
use Filament\Tables\Filters\QueryBuilder;

QueryBuilder::make()
    ->constraintPickerColumns(2)
    ->constraints([
        // ...
    ])
```

This method can be used in a couple of different ways:

- You can pass an integer like `constraintPickerColumns(2)`. This integer is the number of columns used on the `lg` breakpoint and higher. All smaller devices will have just 1 column.
- You can pass an array, where the key is the breakpoint and the value is the number of columns. For example, `constraintPickerColumns(['md' => 2, 'xl' => 4])` will create a 2 column layout on medium devices, and a 4 column layout on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

Breakpoints (`sm`, `md`, `lg`, `xl`, `2xl`) are defined by Tailwind, and can be found in the [Tailwind documentation](#).

Increasing the width of the constraint picker

When you increase the number of columns, the width of the dropdown should increase incrementally to handle the additional columns. If you'd like more control, you can manually set a maximum width for the dropdown using the `constraintPickerWidth()` method. Options correspond to [Tailwind's max-width scale](#). The options are `xs`, `sm`, `md`, `lg`, `xl`, `2xl`, `3xl`, `4xl`, `5xl`, `6xl`, `7xl`:

```
use Filament\Tables\Filters\QueryBuilder;

QueryBuilder::make()
->constraintPickerColumns(3)
->constraintPickerWidth('2xl')
->constraints([
    // ...
])
```

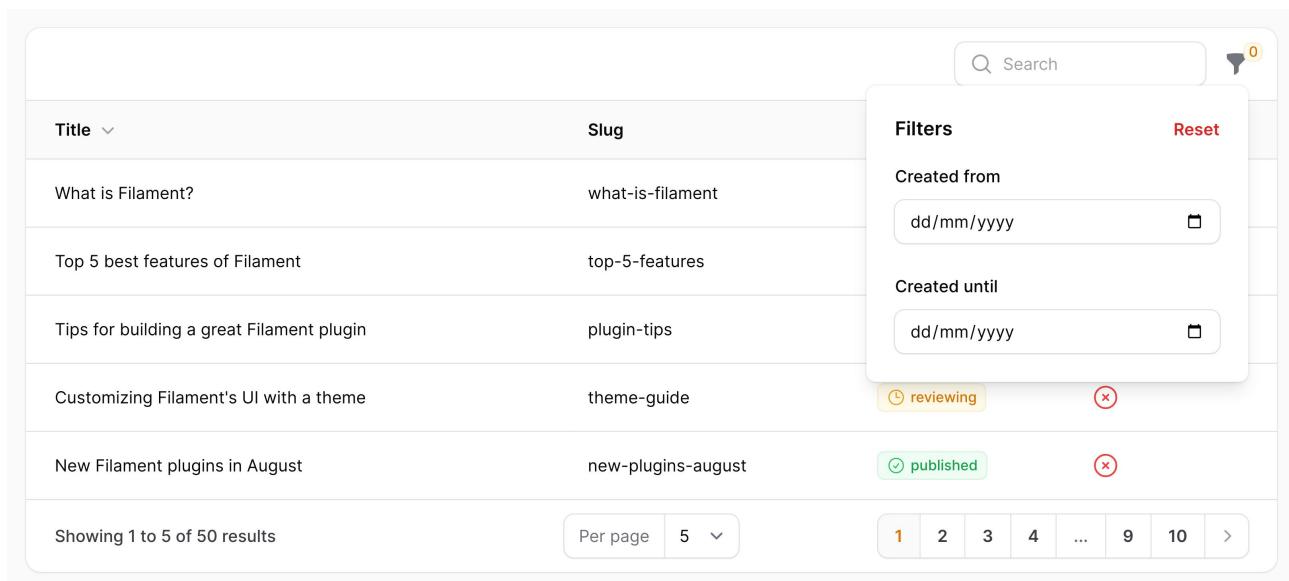
Custom

Custom filter forms

You may use components from the [Form Builder](#) to create custom filter forms. The data from the custom filter form is available in the `$data` array of the `query()` callback:

```
use Filament\Forms\Components\DatePicker;
use Filament\Tables\Filters\Filter;
use Illuminate\Database\Eloquent\Builder;

Filter::make('created_at')
->form([
    DatePicker::make('created_from'),
    DatePicker::make('created_until'),
])
->query(function (Builder $query, array $data): Builder {
    return $query
        ->when(
            $data['created_from'],
            fn (Builder $query, $date): Builder => $query->whereDate('created_at', '>=', $date),
        )
        ->when(
            $data['created_until'],
            fn (Builder $query, $date): Builder => $query->whereDate('created_at', '<=',
                $date),
        );
})
```



The screenshot shows a table component displaying a list of blog posts. The columns are 'Title' and 'Slug'. To the right of the table is a sidebar filter panel. The filter panel includes a search bar, a 'Filters' button, and a 'Reset' button. It has two date range inputs labeled 'Created from' and 'Created until', each with a clear button. Below these are two status filters: 'reviewing' (with a yellow circle icon) and 'published' (with a green circle icon). At the bottom of the sidebar are pagination controls showing page 1 of 10.

Title	Slug	
What is Filament?	what-is-filament	
Top 5 best features of Filament	top-5-features	
Tips for building a great Filament plugin	plugin-tips	
Customizing Filament's UI with a theme	theme-guide	
New Filament plugins in August	new-plugins-august	(reviewing) (published)

Showing 1 to 5 of 50 results

Per page: 5 < >

1 2 3 4 ... 9 10 >

Setting default values for custom filter fields

To customize the default value of a field in a custom filter form, you may use the `default()` method:

```
use Filament\Forms\Components\DatePicker;
use Filament\Tables\Filters\Filter;

Filter::make('created_at')
    ->form([
        DatePicker::make('created_from'),
        DatePicker::make('created_until')
            ->default(now()),
    ])
])
```

Active indicators

When a filter is active, an indicator is displayed above the table content to signal that the table query has been scoped.

The screenshot shows a Filament table interface. At the top, there is a search bar and a badge indicating 2 active filters. Below the header, there are two filter buttons: "Posted by administrator" and "Less than 1 year old". The table has four columns: Title, Slug, Status, and Is featured. The data rows are as follows:

Title	Slug	Status	Is featured
What is Filament?	what-is-filament	published	✓
Top 5 best features of Filament	top-5-features	reviewing	✗
Tips for building a great Filament plugin	plugin-tips	draft	✓
Customizing Filament's UI with a theme	theme-guide	reviewing	✗
New Filament plugins in August	new-plugins-august	published	✗

At the bottom, it says "Showing 1 to 5 of 50 results" and includes a page navigation section with buttons for 1, 2, 3, 4, ..., 9, 10, >.

By default, the label of the filter is used as the indicator. You can override this using the `indicator()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_admin')
    ->label('Administrators only?')
    ->indicator('Administrators')
```

If you are using a [custom filter form](#), you should use `indicateUsing()` to display an active indicator.

Please note: if you do not have an indicator for your filter, then the badge-count of how many filters are active in the table will not include that filter.

Custom active indicators

Not all indicators are simple, so you may need to use `indicateUsing()` to customize which indicators should be shown at any time.

For example, if you have a custom date filter, you may create a custom indicator that formats the selected date:

```

use Carbon\Carbon;
use Filament\Forms\Components\DatePicker;
use Filament\Tables\Filters\Filter;

Filter::make('created_at')
    ->form([DatePicker::make('date')])
    // ...
    ->indicateUsing(function (array $data): ?string {
        if (! $data['date']) {
            return null;
        }

        return 'Created at ' . Carbon::parse($data['date'])->toFormattedDateString();
    })
}

```

Multiple active indicators

You may even render multiple indicators at once, by returning an array of `Indicator` objects. If you have different fields associated with different indicators, you should set the field using the `removeField()` method on the `Indicator` object to ensure that the correct field is reset when the filter is removed:

```

use Carbon\Carbon;
use Filament\Forms\Components\DatePicker;
use Filament\Tables\Filters\Filter;
use Filament\Tables\Filters\Indicator;

Filter::make('created_at')
    ->form([
        DatePicker::make('from'),
        DatePicker::make('until'),
    ])
    // ...
    ->indicateUsing(function (array $data): array {
        $indicators = [];

        if ($data['from'] ?? null) {
            $indicators[] = Indicator::make('Created from ' . Carbon::parse($data['from'])->toFormattedDateString())
                ->removeField('from');
        }

        if ($data['until'] ?? null) {
            $indicators[] = Indicator::make('Created until ' . Carbon::parse($data['until'])->toFormattedDateString())
                ->removeField('until');
        }

        return $indicators;
    })
}

```

Preventing indicators from being removed

You can prevent users from removing an indicator using `removable(false)` on an `Indicator` object:

```
use Carbon\Carbon;
use Filament\Tables\Filters\Indicator;

Indicator::make('Created from ' . Carbon::parse($data['from'])->toFormattedDateString())
->removable(false)
```

Layout

Positioning filters into grid columns

To change the number of columns that filters may occupy, you may use the `filtersFormColumns()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->filtersFormColumns(3);
}
```

Controlling the width of the filters dropdown

To customize the dropdown width, you may use the `filtersFormWidth()` method, and specify a width - `ExtraSmall`, `Small`, `Medium`, `Large`, `ExtraLarge`, `TwoExtraLarge`, `ThreeExtraLarge`, `FourExtraLarge`, `FiveExtraLarge`, `SixExtraLarge` or `SevenExtraLarge`. By default, the width is `ExtraSmall`:

```
use Filament\Support\Facades\MaxWidth;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->filtersFormWidth(MaxWidth::FourExtraLarge);
}
```

Controlling the maximum height of the filters dropdown

To add a maximum height to the filters' dropdown content, so that they scroll, you may use the `filtersFormMaxHeight()` method, passing a [CSS length](#):

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->filtersFormMaxHeight('400px');
}
```

Displaying filters in a modal

To render the filters in a modal instead of in a dropdown, you may use:

```
use Filament\Tables\Enums\FiltersLayout;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ], layout: FiltersLayout::Modal);
}
```

You may use the [trigger action API](#) to [customize the modal](#), including [using a `slideOver\(\)`](#).

Displaying filters above the table content

To render the filters above the table content instead of in a dropdown, you may use:

```
use Filament\Tables\Enums\FiltersLayout;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ], layout: FiltersLayout::AboveContent);
}
```

Filters				Reset
Status	Author	Created from	Created until	
All	All	dd/mm/yyyy	dd/mm/yyyy	
<input type="button" value="Search"/>				
Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	
Top 5 best features of Filament	top-5-features	reviewing	✗	
Tips for building a great Filament plugin	plugin-tips	draft	✓	
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	
New Filament plugins in August	new-plugins-august	published	✗	
Showing 1 to 5 of 50 results		Per page	5	1 2 3 4 ... 9 10 >

Allowing filters above the table content to be collapsed

To allow the filters above the table content to be collapsed, you may use:

```
use Filament\Tables\Enums\FiltersLayout;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ], layout: FiltersLayout::AboveContentCollapsible);
}
```

Displaying filters below the table content

To render the filters below the table content instead of in a dropdown, you may use:

```
use Filament\Tables\Enums\FiltersLayout;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ], layout: FiltersLayout::BelowContent);
}
```

Title	Slug	Status	Is featured
What is Filament?	what-is-filament	published	✓
Top 5 best features of Filament	top-5-features	reviewing	✗
Tips for building a great Filament plugin	plugin-tips	draft	✓
Customizing Filament's UI with a theme	theme-guide	reviewing	✗
New Filament plugins in August	new-plugins-august	published	✗

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Filters Reset

Status	Author	Created from	Created until
All	All	dd/mm/yyyy	dd/mm/yyyy

Hiding the filter indicators

To hide the active filters indicators above the table, you may use `hiddenFilterIndicators()`:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ])
        ->hiddenFilterIndicators();
}
```

Customizing the filter form schema

You may customize the form schema of the entire filter form at once, in order to rearrange filters into your desired layout, and use any of the layout components available to forms. To do this, use the `filterFormSchema()` method, passing a closure function that receives the array of defined `$filters` that you can insert:

```

use Filament\Forms\Components\Section;
use Filament\Tables\Filters\Filter;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->filters([
            Filter::make('is_featured'),
            Filter::make('published_at'),
            Filter::make('author'),
        ])
        ->filtersFormColumns(2)
        ->filtersFormSchema(fn (array $filters): array => [
            Section::make('Visibility')
                ->description('These filters affect the visibility of the records in the
table.')
                ->schema([
                    $filters['is_featured'],
                    $filters['published_at'],
                ])
                ->columns(2)
                ->columnSpanFull(),
                $filters['author'],
            ]);
}

```

In this example, we have put two of the filters inside a `section` component, and used the `columns()` method to specify that the section should have two columns. We have also used the `columnSpanFull()` method to specify that the section should span the full width of the filter form, which is also 2 columns wide. We have inserted each filter into the form schema by using the filter's name as the key in the `$filters` array.

Actions

Overview

Filament's tables can use [Actions](#). They are buttons that can be added to the [end of any table row](#), or even in the [header](#) of a table. For instance, you may want an action to "create" a new record in the header, and then "edit" and "delete" actions on each row. [Bulk actions](#) can be used to execute code when records in the table are selected. Additionally, actions can be added to any [table column](#), such that each cell in that column is a trigger for your action.

It's highly advised that you read the documentation about [customizing action trigger buttons](#) and [action modals](#) to that you are aware of the full capabilities of actions.

Row actions

Action buttons can be rendered at the end of each table row. You can put them in the `[$table->actions ()]` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            // ...
        ]);
}
```

Actions may be created using the static `make ()` method, passing its unique name.

You can then pass a function to `action ()` which executes the task, or a function to `url ()` which creates a link:

```
use App\Models\Post;
use Filament\Tables\Actions\Action;

Action::make('edit')
    ->url(fn (Post $record): string => route('posts.edit', $record))
    ->openUrlInNewTab()

Action::make('delete')
    ->requiresConfirmation()
    ->action(fn (Post $record) => $record->delete())
```

All methods on the action accept callback functions, where you can access the current table `$record` that was clicked.

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	✓ published	✓	>Edit Delete
Top 5 best features of Filament	top-5-features	⌚ reviewing	✗	Edit Delete
Tips for building a great Filament plugin	plugin-tips	✍ draft	✓	Edit Delete
Customizing Filament's UI with a theme	theme-guide	⌚ reviewing	✗	Edit Delete
New Filament plugins in August	new-plugins-august	✓ published	✗	Edit Delete

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Positioning row actions before columns

By default, the row actions in your table are rendered in the final cell of each row. You may move them before the columns by using the `position` argument:

```
use Filament\Tables\Enums\ActionsPosition;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            // ...
        ], position: ActionsPosition::BeforeColumns);
}
```

Title	Slug	Status	Is featured
>Edit Delete What is Filament?	what-is-filament	✓ published	✓
>Edit Delete Top 5 best features of Filament	top-5-features	⌚ reviewing	✗
>Edit Delete Tips for building a great Filament plugin	plugin-tips	✍ draft	✓
>Edit Delete Customizing Filament's UI with a theme	theme-guide	⌚ reviewing	✗
>Edit Delete New Filament plugins in August	new-plugins-august	✓ published	✗

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Positioning row actions before the checkbox column

By default, the row actions in your table are rendered in the final cell of each row. You may move them before the checkbox column by using the `position` argument:

```
use Filament\Tables\Enums\ActionsPosition;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            // ...
        ], position: ActionsPosition::BeforeCells);
}
```

The screenshot shows a table component with the following data:

<input type="checkbox"/> Title	Slug	Status	Is featured
<input checked="" type="checkbox"/> Edit <input type="button" value="Delete"/> What is Filament?	what-is-filament	(✓) published	(✓)
<input checked="" type="checkbox"/> Edit <input type="button" value="Delete"/> Top 5 best features of Filament	top-5-features	(⌚) reviewing	(✗)
<input checked="" type="checkbox"/> Edit <input type="button" value="Delete"/> Tips for building a great Filament plugin	plugin-tips	(✍) draft	(✓)
<input checked="" type="checkbox"/> Edit <input type="button" value="Delete"/> Customizing Filament's UI with a theme	theme-guide	(⌚) reviewing	(✗)
<input checked="" type="checkbox"/> Edit <input type="button" value="Delete"/> New Filament plugins in August	new-plugins-august	(✓) published	(✗)

At the bottom, there is a search bar, a "Showing 1 to 5 of 50 results" message, a "Per page" dropdown set to 5, and a page navigation bar with buttons for 1, 2, 3, 4, ..., 9, 10, >.

Accessing the selected table rows

You may want an action to be able to access all the selected rows in the table. Usually, this is done with a [bulk action](#) in the header of the table. However, you may want to do this with a row action, where the selected rows provide context for the action.

For example, you may want to have a row action that copies the row data to all the selected records. To force the table to be selectable, even if there aren't bulk actions defined, you need to use the `selectable()` method. To allow the action to access the selected records, you need to use the `accessSelectedRecords()` method. Then, you can use the `$selectedRecords` parameter in your action to access the selected records:

```

use Filament\Tables\Table;
use Filament\Tables\Actions\Action;
use Illuminate\Database\Eloquent\Collection;
use Illuminate\Database\Eloquent\Model;

public function table(Table $table): Table
{
    return $table
        ->selectable()
        ->actions([
            Action::make('copyToSelected')
                ->accessSelectedRecords()
                ->action(function (Model $record, Collection $selectedRecords) {
                    $selectedRecords->each(
                        fn (Model $selectedRecord) => $selectedRecord->update([
                            'is_active' => $record->is_active,
                        ]),
                    );
                }),
        ]);
}

```

Bulk actions

Tables also support "bulk actions". These can be used when the user selects rows in the table. Traditionally, when rows are selected, a "bulk actions" button appears in the top left corner of the table. When the user clicks this button, they are presented with a dropdown menu of actions to choose from. You can put them in the `$table->bulkActions()` method:

```

use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->bulkActions([
            // ...
        ]);
}

```

Bulk actions may be created using the static `make()` method, passing its unique name. You should then pass a callback to `action()` which executes the task:

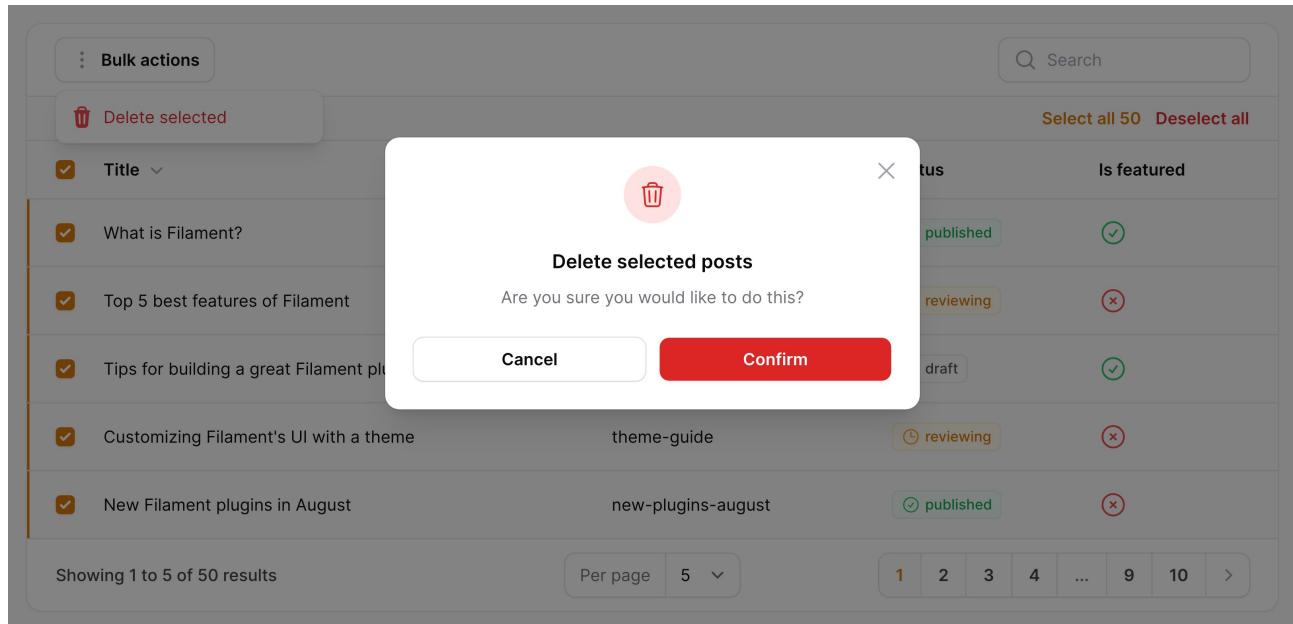
```

use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->requiresConfirmation()
    ->action(fn (Collection $records) => $records->each->delete())

```

The function allows you to access the current table `$records` that are selected. It is an Eloquent collection of models.



Grouping bulk actions

You may use a `BulkActionGroup` object to group multiple bulk actions together in a dropdown. Any bulk actions that remain outside the `BulkActionGroup` will be rendered next to the dropdown's trigger button:

```
use Filament\Tables\Actions\BulkAction;
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->bulkActions([
            BulkActionGroup::make([
                BulkAction::make('delete')
                    ->requiresConfirmation()
                    ->action(fn (Collection $records) => $records->each->delete()),
                BulkAction::make('forceDelete')
                    ->requiresConfirmation()
                    ->action(fn (Collection $records) => $records->each->forceDelete()),
            ]),
            BulkAction::make('export')->button()->action(fn (Collection $records) => ...),
        ]);
}
```

Alternatively, if all of your bulk actions are grouped, you can use the shorthand `groupedBulkActions()` method:

```

use Filament\Tables\Actions\BulkAction;
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groupedBulkActions([
            BulkAction::make('delete')
                ->requiresConfirmation()
                ->action(fn (Collection $records) => $records->each->delete()),
            BulkAction::make('forceDelete')
                ->requiresConfirmation()
                ->action(fn (Collection $records) => $records->each->forceDelete()),
        ]);
}

```

Deselecting records once a bulk action has finished

You may deselect the records after a bulk action has been executed using the `deselectRecordsAfterCompletion()` method:

```

use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->action(fn (Collection $records) => $records->each->delete())
    ->deselectRecordsAfterCompletion()

```

Disabling bulk actions for some rows

You may conditionally disable bulk actions for a specific record:

```

use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Model;

public function table(Table $table): Table
{
    return $table
        ->bulkActions([
            // ...
        ])
        ->checkIfRecordIsSelectableUsing(
            fn (Model $record): bool => $record->status === Status::Enabled,
        );
}

```

Preventing bulk-selection of all pages

The `selectCurrentPageOnly()` method can be used to prevent the user from easily bulk-selecting all records in the table at once, and instead only allows them to select one page at a time:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->bulkActions([
            // ...
        ])
        ->selectCurrentPageOnly();
}
```

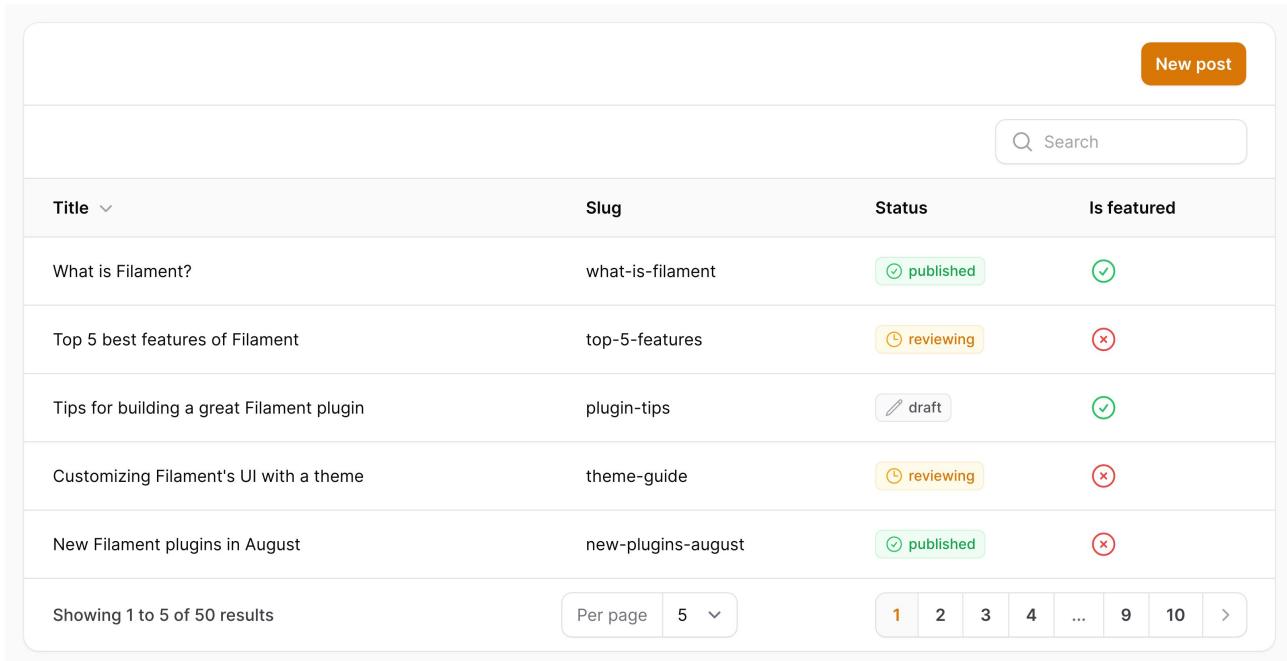
Header actions

Both [row actions](#) and [bulk actions](#) can be rendered in the header of the table. You can put them in the `[$table->headerActions()]` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->headerActions([
            // ...
        ]);
}
```

This is useful for things like "create" actions, which are not related to any specific table row, or bulk actions that need to be more visible.



The screenshot shows a Filament table interface for managing posts. The table has four columns: Title, Slug, Status, and Is featured. The rows represent different posts with the following data:

Title	Slug	Status	Is featured
What is Filament?	what-is-filament	published	✓
Top 5 best features of Filament	top-5-features	reviewing	✗
Tips for building a great Filament plugin	plugin-tips	draft	✓
Customizing Filament's UI with a theme	theme-guide	reviewing	✗
New Filament plugins in August	new-plugins-august	published	✗

At the top right of the table area, there is a "New post" button. Below the table, there is a search bar with the placeholder "Search". At the bottom left, it says "Showing 1 to 5 of 50 results". On the bottom right, there are buttons for "Per page" (set to 5), page navigation (1, 2, 3, 4, ..., 9, 10, >), and a "More" button.

Column actions

Actions can be added to columns, such that when a cell in that column is clicked, it acts as the trigger for an action. You can learn more about [column actions](#) in the documentation.

Prebuilt table actions

Filament includes several prebuilt actions and bulk actions that you can add to a table. Their aim is to simplify the most common Eloquent-related actions:

- [Create](#)
- [Edit](#)
- [View](#)
- [Delete](#)
- [Replicate](#)
- [Force-delete](#)
- [Restore](#)
- [Import](#)
- [Export](#)

Grouping actions

You may use an `ActionGroup` object to group multiple table actions together in a dropdown:

```
use Filament\Tables\Actions\ActionGroup;
use Filament\Tables\Actions\DeleteAction;
use Filament\Tables\Actions>EditAction;
use Filament\Tables\Actions\ViewAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            ActionGroup::make([
                ViewAction::make(),
                EditAction::make(),
                DeleteAction::make(),
            ]),
            // ...
        ]);
}
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing		
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮
Showing 1 to 5 of 50 results		Per page	5	1 2 3 4 ... 9 10 >

Choosing an action group button style

Out of the box, action group triggers have 3 styles - "button", "link", and "icon button".

"Icon button" triggers are circular buttons with an `icon` and no label. Usually, this is the default button style, but you can use it manually with the `iconButton()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])->iconButton()
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing		
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮
Showing 1 to 5 of 50 results		Per page	5	1 2 3 4 ... 9 10 >

"Button" triggers have a background color, label, and optionally an `icon`. You can switch to that style with the `button()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])

->button()
->label('Actions')
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	<input checked="" type="checkbox"/>	Actions
Top 5 best features of Filament	top-5-features	reviewing	<input type="checkbox"/>	View Edit Delete
Tips for building a great Filament plugin	plugin-tips	draft	<input type="checkbox"/>	Actions
Customizing Filament's UI with a theme	theme-guide	reviewing	<input type="checkbox"/>	Actions
New Filament plugins in August	new-plugins-august	published	<input type="checkbox"/>	Actions
Showing 1 to 5 of 50 results		Per page	5	1 2 3 4 ... 9 10 >

"Link" triggers have no background color. They must have a label and optionally an icon. They look like a link that you might find embedded within text. You can switch to that style with the `link()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])

->link()
->label('Actions')
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮ Actions
Top 5 best features of Filament	top-5-features	reviewing		View Edit Delete
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮ Actions
New Filament plugins in August	new-plugins-august	published	✗	⋮ Actions

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Setting the action group button icon

You may set the `icon` of the action group button using the `icon()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])->icon('heroicon-m-ellipsis-horizontal');
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing		View Edit Delete
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮

Showing 1 to 5 of 50 results

Per page: 5

1 2 3 4 ... 9 10 >

Setting the action group button color

You may set the color of the action group button using the `color()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])->color('info');
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing		
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮

Showing 1 to 5 of 50 results

Per page: 5 ▾

1 2 3 4 ... 9 10 >

Setting the action group button size

Buttons come in 3 sizes - `sm`, `md` or `lg`. You may set the size of the action group button using the `size()` method:

```
use Filament\Support\Enums\ActionSize;
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])->size(ActionSize::Small);
```

Title	Slug	Status	Is featured	
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing		
Tips for building a great Filament plugin	plugin-tips	draft		
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮

Showing 1 to 5 of 50 results

Per page: 5 ▾

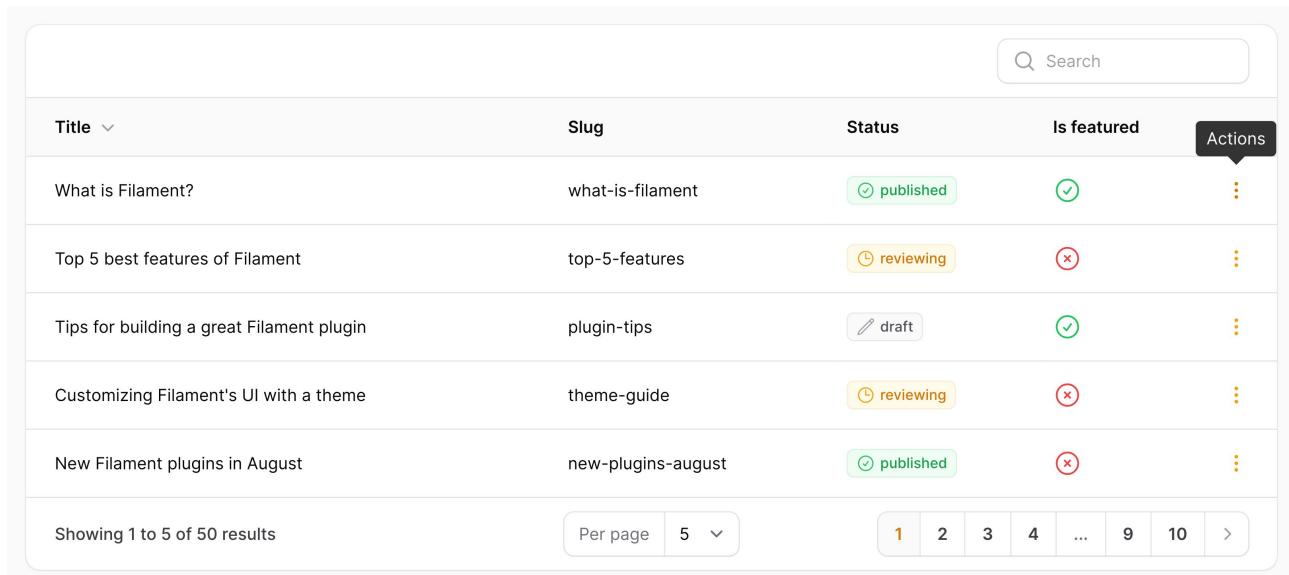
1 2 3 4 ... 9 10 >

Setting the action group tooltip

You may set the tooltip of the action group using the `tooltip()` method:

```
use Filament\Tables\Actions\ActionGroup;

ActionGroup::make([
    // ...
])->tooltip('Actions');
```



The screenshot shows a Filament table interface. At the top right is a search bar with a magnifying glass icon and the placeholder "Search". Below the search bar is a table header with columns: "Title", "Slug", "Status", "Is featured", and "Actions". The "Actions" column contains a dark blue button with a white three-dot menu icon. The table body contains five rows of data. Each row has a "Title" (e.g., "What is Filament?", "Top 5 best features of Filament", etc.), a "Slug" (e.g., "what-is-filament", "top-5-features", etc.), a "Status" badge (e.g., "published", "reviewing", "draft"), an "Is featured" badge (green checkmark or red X), and the dark blue "Actions" button. At the bottom left of the table is the text "Showing 1 to 5 of 50 results". At the bottom center are pagination controls: "Per page" dropdown set to "5", and a numbered navigation bar from 1 to 10.

Title	Slug	Status	Is featured	Actions
What is Filament?	what-is-filament	published	✓	⋮
Top 5 best features of Filament	top-5-features	reviewing	✗	⋮
Tips for building a great Filament plugin	plugin-tips	draft	✓	⋮
Customizing Filament's UI with a theme	theme-guide	reviewing	✗	⋮
New Filament plugins in August	new-plugins-august	published	✗	⋮

Showing 1 to 5 of 50 results

Per page 5 ▾

1 2 3 4 ... 9 10 >

Table action utility injection

All actions, not just table actions, have access to [many utilities](#) within the vast majority of configuration methods. However, in addition to those, table actions have access to a few more:

Injecting the current Eloquent record

If you wish to access the current Eloquent record of the action, define a `$record` parameter:

```
use Illuminate\Database\Eloquent\Model;

function (Model $record) {
    // ...
}
```

Be aware that bulk actions, header actions, and empty state actions do not have access to the `$record`, as they are not related to any table row.

Injecting the current Eloquent model class

If you wish to access the current Eloquent model class of the table, define a `$model` parameter:

```
function (string $model) {
    // ...
}
```

Injecting the current table instance

If you wish to access the current table configuration instance that the action belongs to, define a `$table` parameter:

```
use Filament\Tables\Table;

function (Table $table) {
    // ...
}
```

Layout

The problem with traditional table layouts

Traditional tables are notorious for having bad responsiveness. On mobile, there is only so much flexibility you have when rendering content that is horizontally long:

- Allow the user to scroll horizontally to see more table content
- Hide non-important columns on smaller devices

Both of these are possible with Filament. Tables automatically scroll horizontally when they overflow anyway, and you may choose to show and hide columns based on the responsive [breakpoint](#) of the browser. To do this, you may use a `visibleFrom()` or `hiddenFrom()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('slug')
    ->visibleFrom('md')
```

This is fine, but there is still a glaring issue - **on mobile, the user is unable to see much information in a table row at once without scrolling.**

Thankfully, Filament lets you build responsive table-like interfaces, without touching HTML or CSS. These layouts let you define exactly where content appears in a table row, at each responsive breakpoint.

<input type="checkbox"/> Sort by - ▼				<input type="text"/> Search 🔍	
<input type="checkbox"/>	Dan Harrin	Developer	+1 (555) 555-5555 dan@filamentphp.com	<input checked="" type="checkbox"/> Edit	^
<input type="checkbox"/>	Ryan Chandler	Developer	+1 (555) 555-5555 ryan@filamentphp.com	<input checked="" type="checkbox"/> Edit	^
<input type="checkbox"/>	Zep Fietje	Developer	+1 (555) 555-5555 zep@filamentphp.com	<input checked="" type="checkbox"/> Edit	^
<input type="checkbox"/>	Dennis Koch	Developer	+1 (555) 555-5555 dennis@filamentphp.com	<input checked="" type="checkbox"/> Edit	^
<input type="checkbox"/>	Adam Weston	Developer	+1 (555) 555-5555 adam@filamentphp.com	<input checked="" type="checkbox"/> Edit	^
Showing 1 to 5 of 51 results			Per page	5 ▼	1 2 3 4 ... 10 11 >

Dan Harrin
Developer

Ryan Chandler
Developer

Edit

Allowing columns to stack on mobile

Let's introduce a component - `Split`:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular(),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    TextColumn::make('email'),
])
```

<input type="checkbox"/>		<input type="text"/> Search
<input type="checkbox"/>	Sort by	- <input type="button" value="▼"/>
<input type="checkbox"/>		Dan Harrin
		dan@filamentphp.com
		 Edit
<input type="checkbox"/>		Ryan Chandler
		ryan@filamentphp.com
		 Edit
<input type="checkbox"/>		Zep Fietje
		zep@filamentphp.com
		 Edit
<input type="checkbox"/>		Dennis Koch
		dennis@filamentphp.com
		 Edit
<input type="checkbox"/>		Adam Weston
		adam@filamentphp.com
		 Edit

Showing 1 to 5 of 51 results

Per page

 1 2 3 4 ... 10 11 >

<input type="checkbox"/>		<input type="text"/> Search
<input type="checkbox"/>	Sort by	- <input type="button" value="▼"/>
<input type="checkbox"/>		Dan Harrin
		dan@filamentphp.com
		 Edit
<input type="checkbox"/>		Ryan Chandler
		ryan@filamentphp.com
		 Edit

A `Split` component is used to wrap around columns, and allow them to stack on mobile.

By default, columns within a split will appear aside each other all the time. However, you may choose a responsive `breakpoint` where this behavior starts `from()`. Before this point, the columns will stack on top of each other:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular(),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    TextColumn::make('email'),
])->from('md')
```

In this example, the columns will only appear horizontally aside each other from `md` breakpoint devices onwards:

			<input type="text"/> Search
	<input type="checkbox"/>	Sort by	- ▾
<input type="checkbox"/>		Dan Harrin	dan@filamentphp.com 
<input type="checkbox"/>		Ryan Chandler	ryan@filamentphp.com 
<input type="checkbox"/>		Zep Fietje	zep@filamentphp.com 
<input type="checkbox"/>		Dennis Koch	dennis@filamentphp.com 
<input type="checkbox"/>		Adam Weston	adam@filamentphp.com 
Showing 1 to 5 of 51 results		Per page	5 ▾
1 2 3 4 ... 10 11 >			

A screenshot of a Filament 3.x application interface. At the top right is a search bar with a magnifying glass icon and the placeholder text "Search". Below it is a "Sort by" dropdown menu with a minus sign and a downward arrow. To the left of the search bar is a small square checkbox. The main content area displays a user profile card. It features a circular profile picture of a man, the name "Dan Harrin" in bold black text, the email "dan@filamentphp.com" in regular black text, and an orange "Edit" button with a pencil icon. There is also a small square checkbox to the left of the name.

Preventing a column from creating whitespace

Splits, like table columns, will automatically adjust their whitespace to ensure that each column has proportionate separation. You can prevent this from happening, using `grow(false)`. In this example, we will make sure that the avatar image will sit tightly against the name column:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular()
        ->grow(false),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    TextColumn::make('email'),
])
```

The other columns which are allowed to `grow()` will adjust to consume the newly-freed space:

			<input type="text"/> Search
<input type="checkbox"/>	Sort by	-	<input type="button"/>
<input type="checkbox"/>	 Dan Harrin	dan@filamentphp.com	 Edit
<input type="checkbox"/>	 Ryan Chandler	ryan@filamentphp.com	 Edit
<input type="checkbox"/>	 Zep Fietje	zep@filamentphp.com	 Edit
<input type="checkbox"/>	 Dennis Koch	dennis@filamentphp.com	 Edit
<input type="checkbox"/>	 Adam Weston	adam@filamentphp.com	 Edit
Showing 1 to 5 of 51 results		Per page	5 <input type="button"/>
<input type="button"/> 1 2 3 4 ... 10 11 >			

			<input type="text"/> Search
<input type="checkbox"/>	Sort by	-	<input type="button"/>
<input type="checkbox"/>	 Dan Harrin	dan@filamentphp.com	 Edit
<input type="checkbox"/>			

Stacking within a split

Inside a split, you may stack multiple columns on top of each other vertically. This allows you to display more data inside fewer columns on desktop:

```

use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular(),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    Stack::make([
        TextColumn::make('phone')
            ->icon('heroicon-m-phone'),
        TextColumn::make('email')
            ->icon('heroicon-m-envelope'),
    ]),
])

```

					<input type="text"/> Search
<input type="checkbox"/>	Sort by	-	<	>	
<input type="checkbox"/>	 Dan Harrin	 +1 (555) 555-5555  dan@filamentphp.com			
<input type="checkbox"/>	 Ryan Chandler	 +1 (555) 555-5555  ryan@filamentphp.com			
<input type="checkbox"/>	 Zep Fietje	 +1 (555) 555-5555  zep@filamentphp.com			
<input type="checkbox"/>	 Dennis Koch	 +1 (555) 555-5555  dennis@filamentphp.com			
<input type="checkbox"/>	 Adam Weston	 +1 (555) 555-5555  adam@filamentphp.com			
Showing 1 to 5 of 51 results		Per page	5	<	>
		1	2	3	4
		...	10	11	>

Hiding a stack on mobile

Similar to individual columns, you may choose to hide a stack based on the responsive [breakpoint](#) of the browser. To do this, you may use a `visibleFrom()` method:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular(),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    Stack::make([
        TextColumn::make('phone')
            ->icon('heroicon-m-phone'),
        TextColumn::make('email')
            ->icon('heroicon-m-envelope'),
    ]) ->visibleFrom('md'),
])
```

		<input type="text"/> Search
<input type="checkbox"/>	Sort by	- <input type="button" value="▼"/>
<input type="checkbox"/>	 Dan Harrin	 +1 (555) 555-5555  dan@filamentphp.com 
<input type="checkbox"/>	 Ryan Chandler	 +1 (555) 555-5555  ryan@filamentphp.com 
<input type="checkbox"/>	 Zep Fietje	 +1 (555) 555-5555  zep@filamentphp.com 
<input type="checkbox"/>	 Dennis Koch	 +1 (555) 555-5555  dennis@filamentphp.com 
<input type="checkbox"/>	 Adam Weston	 +1 (555) 555-5555  adam@filamentphp.com 

Showing 1 to 5 of 51 results

Per page ...

		<input type="text"/> Search
<input type="checkbox"/>	Sort by	- <input type="button" value="▼"/>
<input type="checkbox"/>	 Dan Harrin	
<input type="checkbox"/>		

Aligning stacked content

By default, columns within a stack are aligned to the start. You may choose to align columns within a stack to the `Alignment::Center` or `Alignment::End`:

```

use Filament\Support\Enums\Alignment;
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')
        ->circular(),
    TextColumn::make('name')
        ->weight(FontWeight::Bold)
        ->searchable()
        ->sortable(),
    Stack::make([
        TextColumn::make('phone')
            ->icon('heroicon-m-phone')
            ->grow(false),
        TextColumn::make('email')
            ->icon('heroicon-m-envelope')
            ->grow(false),
    ])
    ->alignment(Alignment::End)
    ->visibleFrom('md'),
])
]
)

```

Ensure that the columns within the stack have `grow(false)` set, otherwise they will stretch to fill the entire width of the stack and follow their own alignment configuration instead of the stack's.

<input type="checkbox"/>	Dan Harrin	+1 (555) 555-5555 dan@filamentphp.com		
<input type="checkbox"/>	Ryan Chandler	+1 (555) 555-5555 ryan@filamentphp.com		
<input type="checkbox"/>	Zep Fietje	+1 (555) 555-5555 zep@filamentphp.com		
<input type="checkbox"/>	Dennis Koch	+1 (555) 555-5555 dennis@filamentphp.com		
<input type="checkbox"/>	Adam Weston	+1 (555) 555-5555 adam@filamentphp.com		

Showing 1 to 5 of 51 results

Per page: 5

1 2 3 4 ... 10 11 >

Spacing stacked content

By default, stacked content has no vertical padding between columns. To add some, you may use the `space()` method, which accepts either `1`, `2`, or `3`, corresponding to [Tailwind's spacing scale](#):

```
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\TextColumn;

Stack::make([
    TextColumn::make('phone')
        ->icon('heroicon-m-phone'),
    TextColumn::make('email')
        ->icon('heroicon-m-envelope'),
])->space(1)
```

Controlling column width using a grid

Sometimes, using a `Split` creates inconsistent widths when columns contain lots of content. This is because it's powered by Flexbox internally and each row individually controls how much space is allocated to content.

Instead, you may use a `Grid` layout, which uses CSS Grid Layout to allow you to control column widths:

```
use Filament\Tables\Columns\Layout\Grid;
use Filament\Tables\Columns\TextColumn;

Grid::make([
    'lg' => 2,
])
->schema([
    TextColumn::make('phone')
        ->icon('heroicon-m-phone'),
    TextColumn::make('email')
        ->icon('heroicon-m-envelope'),
])
```

These columns will always consume equal width within the grid, from the `lg` breakpoint.

You may choose to customize the number of columns within the grid at other breakpoints:

```
use Filament\Tables\Columns\Layout\Grid;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\TextColumn;

Grid::make([
    'lg' => 2,
    '2xl' => 4,
])
->schema([
    Stack::make([
        TextColumn::make('name'),
        TextColumn::make('job'),
    ]),
    TextColumn::make('phone')
        ->icon('heroicon-m-phone'),
    TextColumn::make('email')
        ->icon('heroicon-m-envelope'),
])
])
```

And you can even control how many grid columns will be consumed by each component at each [breakpoint](#):

```
use Filament\Tables\Columns\Layout\Grid;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\TextColumn;

Grid::make([
    'lg' => 2,
    '2xl' => 5,
])
->schema([
    Stack::make([
        TextColumn::make('name'),
        TextColumn::make('job'),
    ])->columnSpan([
        'lg' => 'full',
        '2xl' => 2,
    ]),
    TextColumn::make('phone')
        ->icon('heroicon-m-phone')
        ->columnSpan([
            '2xl' => 2,
        ]),
    TextColumn::make('email')
        ->icon('heroicon-m-envelope'),
])
])
```

Collapsible content

When you're using a column layout like split or stack, then you can also add collapsible content. This is very useful for when you don't want to display all data in the table at once, but still want it to be accessible to the user if they need to access it, without navigating away.

Split and stack components can be made `collapsible()`, but there is also a dedicated `Panel` component that provides a pre-styled background color and border radius, to separate the collapsible content from the rest:

```

use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Panel;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

[
    Split::make([
        ImageColumn::make('avatar')
            ->circular(),
        TextColumn::make('name')
            ->weight(FontWeight::Bold)
            ->searchable()
            ->sortable(),
    ]),
    Panel::make([
        Stack::make([
            TextColumn::make('phone')
                ->icon('heroicon-m-phone'),
            TextColumn::make('email')
                ->icon('heroicon-m-envelope'),
        ]),
    ]) ->collapsible(),
]

```

You can expand a panel by default using the `collapsed(false)` method:

```

use Filament\Tables\Columns\Layout\Panel;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\TextColumn;

Panel::make([
    Split::make([
        TextColumn::make('phone')
            ->icon('heroicon-m-phone'),
        TextColumn::make('email')
            ->icon('heroicon-m-envelope'),
    ]) ->from('md'),
]) ->collapsed(false)

```

The screenshot shows a Filament application interface with a search bar at the top right. Below it is a sorting dropdown labeled "Sort by" with options "- >". A list of five contacts is displayed in a grid:

- Dan Harrin**: Contact info: +1 (555) 555-5555, dan@filamentphp.com. Edit button.
- Ryan Chandler**: Contact info: +1 (555) 555-5555, dan@filamentphp.com. Edit button.
- Zep Fietje**: Contact info: +1 (555) 555-5555, dan@filamentphp.com. Edit button.
- Dennis Koch**: Contact info: +1 (555) 555-5555, dan@filamentphp.com. Edit button.
- Adam Weston**: Contact info: +1 (555) 555-5555, dan@filamentphp.com. Edit button.

At the bottom, it says "Showing 1 to 5 of 51 results" with a "Per page" dropdown set to 5, and a pagination bar showing pages 1 through 11.

The screenshot shows a detailed view of a contact record for **Dan Harrin**. At the top is a search bar. Below it is a sorting dropdown labeled "Sort by" with options "- >". The contact information is shown in a card:

- Dan Harrin**
- +1 (555) 555-5555**
- dan@filamentphp.com**

An orange "Edit" button is located below the card.

Arranging records into a grid

Sometimes, you may find that your data fits into a grid format better than a list. Filament can handle that too!

Simply use the `$table->contentGrid()` method:

```
use Filament\Tables\Columns\Layout\Stack;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            Stack::make([
                // Columns
            ]),
        ])
        ->contentGrid([
            'md' => 2,
            'xl' => 3,
        ]);
}
```

In this example, the rows will be displayed in a grid:

- On mobile, they will be displayed in 1 column only.
- From the `md` breakpoint, they will be displayed in 2 columns.
- From the `xl` breakpoint onwards, they will be displayed in 3 columns.

These settings are fully customizable, any breakpoint from `sm` to `2xl` can contain `1` to `12` columns.

The screenshot shows a table component with the following data:

	Developer	Action
<input type="checkbox"/>	Dan Harrin Developer	<input checked="" type="button"/> Edit
<input type="checkbox"/>	Ryan Chandler Developer	<input checked="" type="button"/> Edit
<input type="checkbox"/>	Zep Fietje Developer	<input checked="" type="button"/> Edit
<input type="checkbox"/>	Dennis Koch Developer	<input checked="" type="button"/> Edit
<input type="checkbox"/>	Adam Weston Developer	<input checked="" type="button"/> Edit
<input type="checkbox"/>	Ryan Scherler Developer	<input checked="" type="button"/> Edit

At the bottom, it says "Showing 1 to 6 of 51 results" and has a navigation bar with pages 1 through 9.

The screenshot shows a table component with two rows. The first row contains a user profile picture, the name "Dan Harrin", the title "Developer", and an "Edit" button. The second row contains a user profile picture, the name "Ryan Chandler", and an "Edit" button. Each row has a collapse icon (triangle) on the right side.

Custom HTML

You may add custom HTML to your table using a `View` component. It can even be `collapsible()`:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\View;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

[
    Split::make([
        ImageColumn::make('avatar')
            ->circular(),
        TextColumn::make('name')
            ->weight(FontWeight::Bold)
            ->searchable()
            ->sortable(),
    ]),
    View::make('users.table.collapsible-row-content')
        ->collapsible(),
]
```

Now, create a `/resources/views/users/table/collapsible-row-content.blade.php` file, and add in your HTML. You can access the table record using `$getRecord()`:

```
<p class="px-4 py-3 bg-gray-100 rounded-lg">
    <span class="font-medium">
        Email address:
    </span>

    <span>
        {{ $getRecord() ->email }}
    </span>
</p>
```

Embedding other components

You could even pass in columns or other layout components to the `components()` method:

```
use Filament\Support\Enums\FontWeight;
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\Layout\View;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

[
    Split::make([
        ImageColumn::make('avatar')
            ->circular(),
        TextColumn::make('name')
            ->weight(FontWeight::Bold)
            ->searchable()
            ->sortable(),
    ]),
    View::make('users.table.collapsible-row-content')
        ->components([
            TextColumn::make('email')
                ->icon('heroicon-m-envelope'),
        ])
        ->collapsible(),
]
```

Now, render the components in the Blade file:

```
<div class="px-4 py-3 bg-gray-100 rounded-lg">
    <x-filament-tables::columns.layout
        :components="$getComponents()"
        :record="$getRecord()"
        :record-key="$recordKey"
    />
</div>
```

Summaries

Overview

You may render a "summary" section below your table content. This is great for displaying the results of calculations such as averages, sums, counts, and ranges of the data in your table.

By default, there will be a single summary line for the current page of data, and an additional summary line for the totals for all data if multiple pages are available. You may also add summaries for groups of records, see "["Summarising groups of rows"](#)".

"Summarizer" objects can be added to any table column using the `summarize()` method:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
->summarize(Average::make())
```

Multiple "summarizers" may be added to the same column:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
->numeric()
->summarize([
    Average::make(),
    Range::make(),
])
```

The first column in a table may not use summarizers. That column is used to render the heading and subheading/s of the summary section.

					<input type="text"/> Search						
Title	Slug	Rating	Status	Is featured							
What is Filament?	what-is-filament	8.1	✓ published	✓							
Top 5 best features of Filament	top-5-features	9.3	⌚ reviewing	✗							
Tips for building a great Filament plugin	plugin-tips	9.7	✍ draft	✓							
Customizing Filament's UI with a theme	theme-guide	9.5	⌚ reviewing	✗							
New Filament plugins in August	new-plugins-august	8.4	✓ published	✗							
Summary		Rating	Is featured								
		Average									
This page		9	Count								
		8.1 - 9.7									
All posts		Average	Count								
		6.06	24								
		1 - 10									
Showing 1 to 5 of 50 results		Per page	5 ▾								
				1	2	3	4	...	9	10	>

Available summarizers

Filament ships with four types of summarizer:

- [Average](#)
- [Count](#)
- [Range](#)
- [Sum](#)

You may also [create your own custom summarizers](#) to display data in whatever way you wish.

Average

Average can be used to calculate the average of all values in the dataset:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
->summarize(Average::make())
```

In this example, all ratings in the table will be added together and divided by the number of ratings.

Count

Count can be used to find the total number of values in the dataset. Unless you just want to calculate the number of rows, you will probably want to [scope the dataset](#) as well:

```
use Filament\Tables\Columns\IconColumn;
use Filament\Tables\Columns\Summarizers\Count;
use Illuminate\Database\Query\Builder;

IconColumn::make('is_published')
    ->boolean()
    ->summarize(
        Count::make()->query(fn (Builder $query) => $query->where('is_published', true)),
    ),
```

In this example, the table will calculate how many posts are published.

Counting the occurrence of icons

Using a count on an [icon column](#) allows you to use the `icons()` method, which gives the user a visual representation of how many of each icon are in the table:

```
use Filament\Tables\Columns\IconColumn;
use Filament\Tables\Columns\Summarizers\Count;
use Illuminate\Database\Query\Builder;

IconColumn::make('is_published')
    ->boolean()
    ->summarize(Count::make()->icons()),
```

Range

Range can be used to calculate the minimum and maximum value in the dataset:

```
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
    ->summarize(Range::make())
```

In this example, the minimum and maximum price in the table will be found.

Date range

You may format the range as dates using the `minimalDateTimeDifference()` method:

```
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('created_at')
    ->dateTime()
    ->summarize(Range::make()->minimalDateTimeDifference())
```

This method will present the minimal difference between the minimum and maximum date. For example:

- If the minimum and maximum dates are different days, only the dates will be displayed.

- If the minimum and maximum dates are on the same day at different times, both the date and time will be displayed.
- If the minimum and maximum dates and times are identical, they will only appear once.

Text range

You may format the range as text using the `minimalTextualDifference()` method:

```
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('sku')
->summarize(Range::make() ->minimalTextualDifference())
```

This method will present the minimal difference between the minimum and maximum. For example:

- If the minimum and maximum start with different letters, only the first letters will be displayed.
- If the minimum and maximum start with the same letter, more of the text will be rendered until a difference is found.
- If the minimum and maximum are identical, they will only appear once.

Including null values in the range

By default, we will exclude null values from the range. If you would like to include them, you may use the

`excludeNull(false)` method:

```
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('sku')
->summarize(Range::make() ->excludeNull(false))
```

Sum

Sum can be used to calculate the total of all values in the dataset:

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->summarize(Sum::make())
```

In this example, all prices in the table will be added together.

Setting a label

You may set a summarizer's label using the `label()` method:

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->summarize(Sum::make() ->label('Total'))
```

Scoping the dataset

You may apply a database query scope to a summarizer's dataset using the `query()` method:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Query\Builder;

TextColumn::make('rating')
    ->summarize(
        Average::make()->query(fn (Builder $query) => $query->where('is_published', true)),
    ),
)
```

In this example, now only rows where `is_published` is set to `true` will be used to calculate the average.

This feature is especially useful with the `count` summarizer, as it can count how many records in the dataset pass a test:

```
use Filament\Tables\Columns\IconColumn;
use Filament\Tables\Columns\Summarizers\Count;
use Illuminate\Database\Query\Builder;

IconColumn::make('is_published')
    ->boolean()
    ->summarize(
        Count::make()->query(fn (Builder $query) => $query->where('is_published', true)),
    ),
)
```

In this example, the table will calculate how many posts are published.

Formatting

Number formatting

The `numeric()` method allows you to format an entry as a number:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
    ->summarize(Average::make()->numeric())
```

If you would like to customize the number of decimal places used to format the number with, you can use the `decimalPlaces` argument:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
    ->summarize(Average::make()->numeric(
        decimalPlaces: 0,
    ))
```

By default, your app's locale will be used to format the number suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
->summarize(Average::make()->numeric(
    locale: 'nl',
))
```

Alternatively, you can set the default locale used across your app using the `Table::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Tables\Table;

Table::$defaultNumberLocale = 'nl';
```

Currency formatting

The `money()` method allows you to easily format monetary values, in any currency:

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->summarize(Sum::make()->money('EUR'))
```

There is also a `divideBy` argument for `money()` that allows you to divide the original value by a number before formatting it. This could be useful if your database stores the price in cents, for example:

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->summarize(Sum::make()->money('EUR', divideBy: 100))
```

By default, your app's locale will be used to format the money suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')
->summarize(Sum::make()->money('EUR', locale: 'nl'))
```

Alternatively, you can set the default locale used across your app using the `Table::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Tables\Table;

Table::$defaultNumberLocale = 'nl';
```

Limiting text length

You may `limit()` the length of the summary's value:

```
use Filament\Tables\Columns\Summarizers\Range;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('sku')
->summarize(Range::make() ->limit(5))
```

Adding a prefix or suffix

You may add a prefix or suffix to the summary's value:

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Support\HtmlString;

TextColumn::make('volume')
->summarize(Sum::make()
->prefix('Total volume: ')
->suffix(new HtmlString(' m3')))
)
```

Custom summaries

You may create a custom summary by returning the value from the `using()` method:

```
use Filament\Tables\Columns\Summarizers\Summarizer;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Query\Builder;

TextColumn::make('name')
->summarize(Summarizer::make()
->label('First last name')
->using(fn (Builder $query): string => $query->min('last_name')))
```

The callback has access to the database `$query` builder instance to perform calculations with. It should return the value to display in the table.

Conditionally hiding the summary

To hide a summary, you may pass a boolean, or a function that returns a boolean, to the `hidden()` method. If you need it, you can access the Eloquent query builder instance for that summarizer via the `$query` argument of the function:

```
use Filament\Tables\Columns\Summarizers\Summarizer;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('sku')
->summarize(Summarizer::make()
->hidden(fn (Builder $query): bool => ! $query->exists())))
```

Alternatively, you can use the `visible()` method to achieve the opposite effect:

```
use Filament\Tables\Columns\Summarizers\Summarizer;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Builder;

TextColumn::make('sku')
    ->summarize(Summarizer::make())
    ->visible(fn (Builder $query): bool => $query->exists()))
```

Summarising groups of rows

You can use summaries with `groups` to display a summary of the records inside a group. This works automatically if you choose to add a summariser to a column in a grouped table.

Hiding the grouped rows and showing the summary only

You may hide the rows inside groups and just show the summary of each group using the `groupsOnly()` method. This is beneficial in many reporting scenarios.

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            TextColumn::make('views_count')
                ->summarize(Sum::make()),
            TextColumn::make('likes_count')
                ->summarize(Sum::make()),
        ])
        ->defaultGroup('category')
        ->groupsOnly();
}
```

Grouping

Overview

You may allow users to group table rows together using a common attribute. This is useful for displaying lots of data in a more organized way.

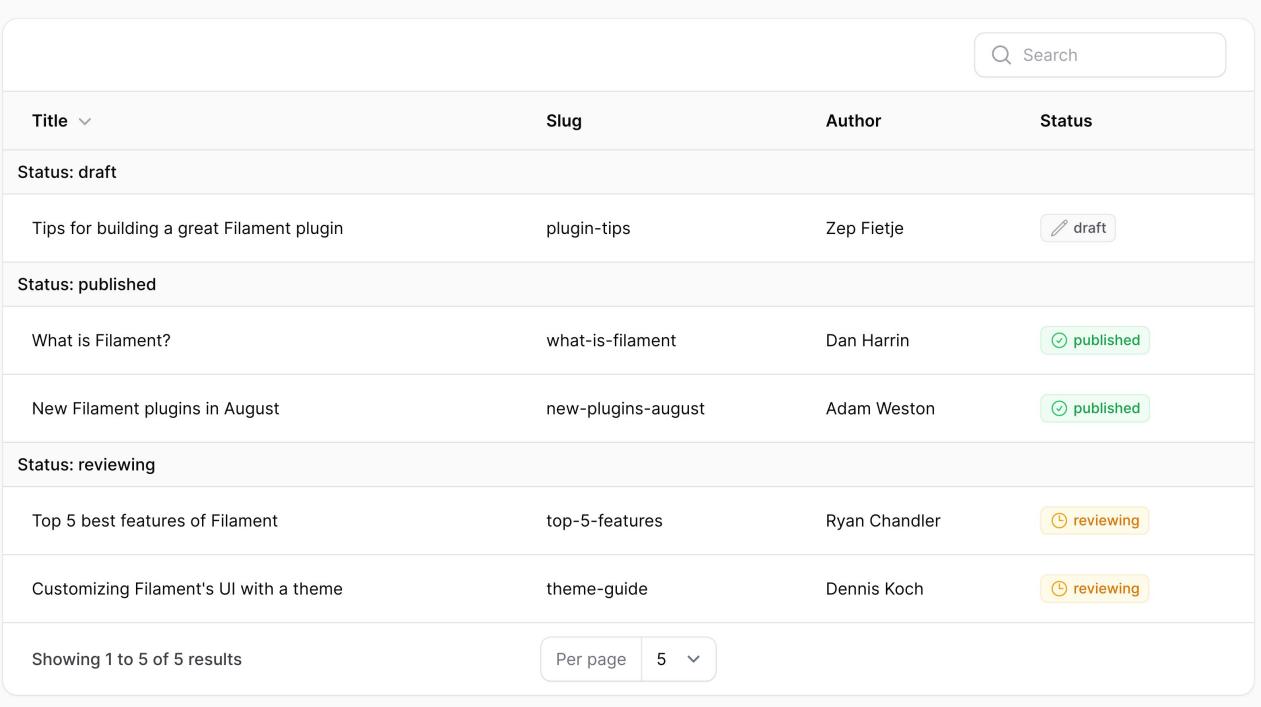
Groups can be set up using the name of the attribute to group by (e.g. `'status'`), or a `Group` object which allows you to customize the behavior of that grouping (e.g. `Group::make('status')->collapsible()`).

Grouping rows by default

You may want to always group posts by a specific attribute. To do this, pass the group to the `defaultGroup()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->defaultGroup('status');
}
```



The screenshot shows a Filament table with the following data:

Title	Slug	Author	Status
Status: draft			
Tips for building a great Filament plugin	plugin-tips	Zep Fietje	✓ draft
Status: published			
What is Filament?	what-is-filament	Dan Harrin	✓ published
New Filament plugins in August	new-plugins-august	Adam Weston	✓ published
Status: reviewing			
Top 5 best features of Filament	top-5-features	Ryan Chandler	⌚ reviewing
Customizing Filament's UI with a theme	theme-guide	Dennis Koch	⌚ reviewing

At the bottom left, it says "Showing 1 to 5 of 5 results". At the bottom right, there are buttons for "Per page" and a dropdown menu showing "5".

Allowing users to choose between groupings

You may also allow users to pick between different groupings, by passing them in an array to the `groups()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            'status',
            'category',
        ]);
}
```

You can use both `groups()` and `defaultGroup()` together to allow users to choose between different groupings, but have a default grouping set:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            'status',
            'category',
        ])
        ->defaultGroup('status');
}
```

Grouping by a relationship attribute

You can also group by a relationship attribute using dot-syntax. For example, if you have an `author` relationship which has a `name` attribute, you can use `author.name` as the name of the attribute:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            'author.name',
        ]);
}
```

Setting a grouping label

By default, the label of the grouping will be generated based on the attribute. You may customize it with a `Group` object, using the `label()` method:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('author.name')
                ->label('Author name'),
        ]);
}
```

Setting a group title

By default, the title of a group will be the value of the attribute. You may customize it by returning a new title from the `getTitleFromRecordUsing()` method of a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->getTitleFromRecordUsing(fn (Post $record): string => ucfirst($record->status->getLabel())),
        ]);
}
```

Disabling the title label prefix

By default, the title is prefixed with the label of the group. To disable this prefix, utilize the `titlePrefixedWithLabel(false)` method:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->titlePrefixedWithLabel(false),
        ]);
}
```

Setting a group description

You may also set a description for a group, which will be displayed underneath the group title. To do this, use the `getDescriptionFromRecordUsing()` method on a `Group` object:

```

use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->getDescriptionFromRecordUsing(fn (Post $record): string => $record->status-
                >getDescription()),
        ]);
}

```

<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10px;"></td><td style="width: 60px; text-align: right;">Search</td><td></td><td></td></tr> </table>					Search		
	Search						
Title	Slug	Author	Status				
Status: draft Posts that are still being written.							
Tips for building a great Filament plugin							
	plugin-tips	Zep Fietje	∅ draft				
Status: published Posts that are public on the website.							
What is Filament?							
	what-is-filament	Dan Harrin	🕒 published				
New Filament plugins in August							
	new-plugins-august	Adam Weston	🕒 published				
Status: reviewing Posts that are being checked by the content team.							
Top 5 best features of Filament							
	top-5-features	Ryan Chandler	🕒 reviewing				
Customizing Filament's UI with a theme							
	theme-guide	Dennis Koch	🕒 reviewing				
Showing 1 to 5 of 5 results		Per page	5 ▾				

Setting a group key

By default, the key of a group will be the value of the attribute. It is used internally as a raw identifier of that group, instead of the [title](#). You may customize it by returning a new key from the `getKeyFromRecordUsing()` method of a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->getKeyFromRecordUsing(fn (Post $record): string => $record->status->value),
        ]);
}
```

Date groups

When using a date-time column as a group, you may want to group by the date only, and ignore the time. To do this, use the `date()` method on a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('created_at')
                ->date(),
        ]);
}
```

Collapsible groups

You can allow rows inside a group to be collapsed underneath their group title. To enable this, use a `Group` object with the `collapsible()` method:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('author.name')
                ->collapsible(),
        ]);
}
```

Summarising groups

You can use `summaries` with groups to display a summary of the records inside a group. This works automatically if you choose to add a summariser to a column in a grouped table.

Hiding the grouped rows and showing the summary only

You may hide the rows inside groups and just show the summary of each group using the `groupsOnly()` method. This is very useful in many reporting scenarios.

```
use Filament\Tables\Columns\Summarizers\Sum;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->columns([
            TextColumn::make('views_count')
                ->summarize(Sum::make()),
            TextColumn::make('likes_count')
                ->summarize(Sum::make()),
        ])
        ->defaultGroup('category')
        ->groupsOnly();
}
```

Customizing the Eloquent query ordering behavior

Some features require the table to be able to order an Eloquent query according to a group. You can customize how we do this using the `orderQueryUsing()` method on a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Builder;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->orderQueryUsing(fn (Builder $query, string $direction) => $query-
                    >orderBy('status', $direction)),
        ]);
}
```

Customizing the Eloquent query scoping behavior

Some features require the table to be able to scope an Eloquent query according to a group. You can customize how we do this using the `scopeQueryByKeyUsing()` method on a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Illuminate\Database\Eloquent\Builder;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->scopeQueryByKeyUsing(fn (Builder $query, string $key) => $query->where('status', $key)),
        ]);
}
```

Customizing the Eloquent query grouping behavior

Some features require the table to be able to group an Eloquent query according to a group. You can customize how we do this using the `groupQueryUsing()` method on a `Group` object:

```
use Filament\Tables\Grouping\Group;
use Illuminate\Database\Eloquent\Builder;

public function table(Table $table): Table
{
    return $table
        ->groups([
            Group::make('status')
                ->groupQueryUsing(fn (Builder $query) => $query->groupBy('status')),
        ]);
}
```

Customizing the groups dropdown trigger action

To customize the groups dropdown trigger button, you may use the `groupRecordsTriggerAction()` method, passing a closure that returns an action. All methods that are available to [customize action trigger buttons](#) can be used:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            // ...
        ])
        ->groupRecordsTriggerAction(
            fn (Action $action) => $action
                ->button()
                ->label('Group records'),
        );
}
```

Using the grouping settings dropdown on desktop

By default, the grouping settings dropdown will only be shown on mobile devices. On desktop devices, the grouping settings are in the header of the table. You can enable the dropdown on desktop devices too by using the `groupingSettingsInDropdownOnDesktop()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->groups([
            // ...
        ])
        ->groupingSettingsInDropdownOnDesktop();
}
```

Hiding the grouping settings

You can hide the grouping settings interface using the `groupingSettingsHidden()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->defaultGroup('status')
        ->groupingSettingsHidden();
}
```

Hiding the grouping direction setting only

You can hide the grouping direction select interface using the `groupingDirectionSettingHidden()` method:

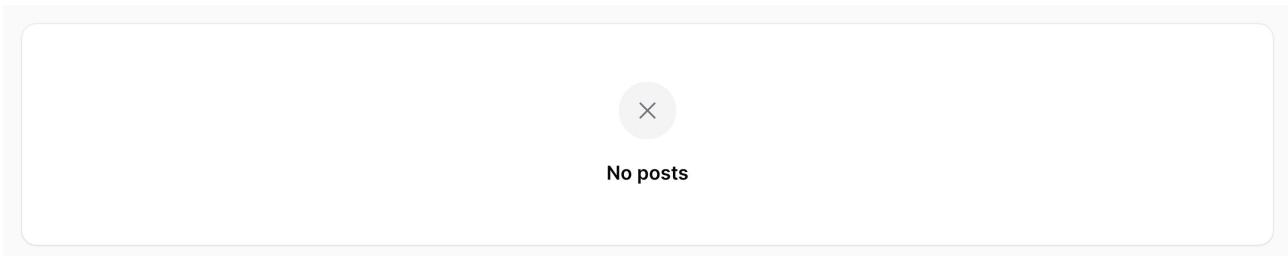
```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->defaultGroup('status')
        ->groupingDirectionSettingHidden();
}
```

Empty State

Overview

The table's "empty state" is rendered when there are no rows in the table.

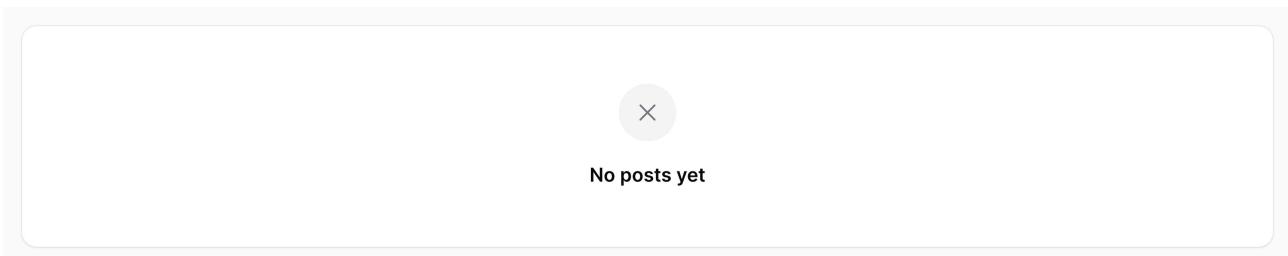


Setting the empty state heading

To customize the heading of the empty state, use the `emptyStateHeading()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->emptyStateHeading('No posts yet');
}
```

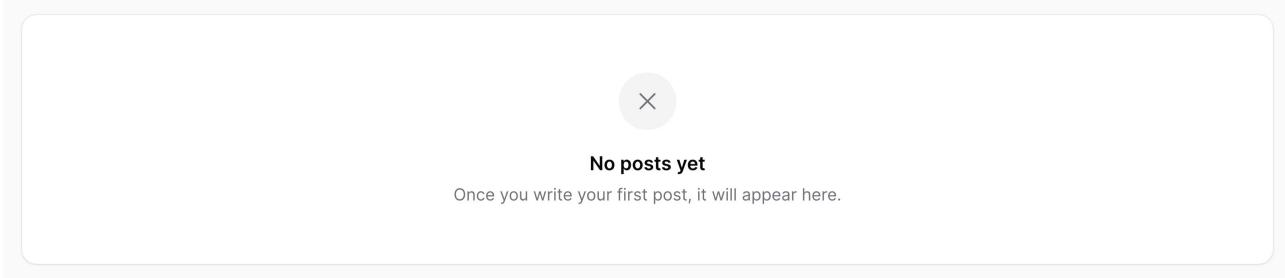


Setting the empty state description

To customize the description of the empty state, use the `emptyStateDescription()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->emptyStateDescription('Once you write your first post, it will appear here.');
}
```

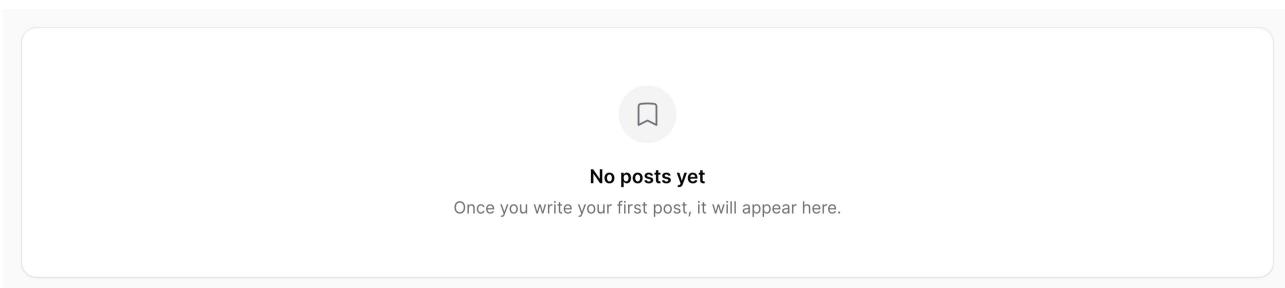


Setting the empty state icon

To customize the icon of the empty state, use the `emptyStateIcon()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->emptyStateIcon('heroicon-o-bookmark');
}
```

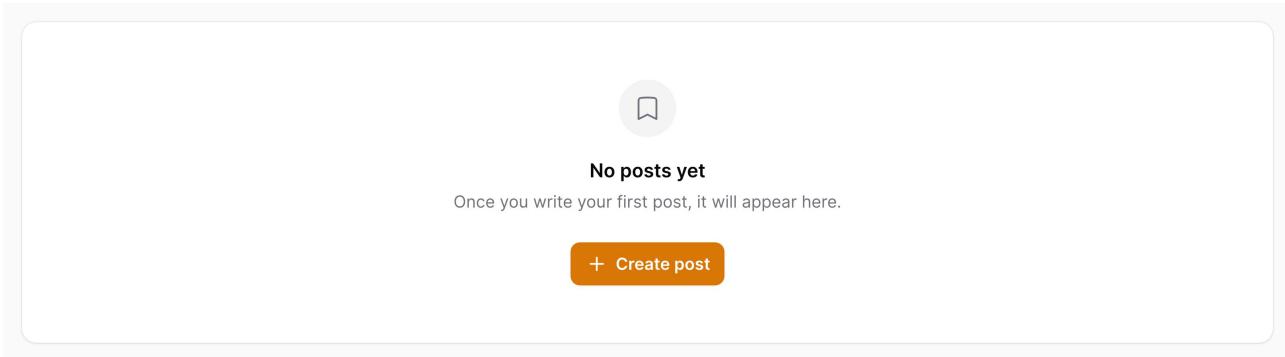


Adding empty state actions

You can add Actions to the empty state to prompt users to take action. Pass these to the `emptyStateActions()` method:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->emptyStateActions([
            Action::make('create')
                ->label('Create post')
                ->url(route('posts.create'))
                ->icon('heroicon-m-plus')
                ->button(),
        ]);
}
```



Using a custom empty state view

You may use a completely custom empty state view by passing it to the `emptyState()` method:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->emptyState(view('tables.posts.empty-state'));
}
```

Advanced

Pagination

Disabling pagination

By default, tables will be paginated. To disable this, you should use the `$table->paginated(false)` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->paginated(false);
}
```

Customizing the pagination options

You may customize the options for the paginated records per page select by passing them to the `paginated()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->paginated([10, 25, 50, 100, 'all']);
}
```

Customizing the default pagination page option

To customize the default number of records shown use the `defaultPaginationPageOption()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->defaultPaginationPageOption(25);
}
```

Preventing query string conflicts with the pagination page

By default, Livewire stores the pagination state in a `page` parameter of the URL query string. If you have multiple tables on the same page, this will mean that the pagination state of one table may be overwritten by the state of another table.

To fix this, you may define a `$table->queryStringIdentifier()`, to return a unique query string identifier for that table:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->queryStringIdentifier('users');
}
```

Displaying links to the first and the last pagination page

To add "extreme" links to the first and the last page using the `extremePaginationLinks()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->extremePaginationLinks();
}
```

Using simple pagination

You may use simple pagination by overriding `paginateTableQuery()` method.

First, locate your Livewire component. If you're using a resource from the Panel Builder and you want to add simple pagination to the List page, you'll want to open the `Pages/List.php` file in the resource, not the resource class itself.

```
use Illuminate\Contracts\Pagination\Paginator;
use Illuminate\Database\Eloquent\Builder;

protected function paginateTableQuery(Builder $query): Paginator
{
    return $query->simplePaginate(($this->getTableRecordsPerPage() === 'all') ? $query->count()
: $this->getTableRecordsPerPage());
}
```

Using cursor pagination

You may use cursor pagination by overriding `paginateTableQuery()` method.

First, locate your Livewire component. If you're using a resource from the Panel Builder and you want to add simple pagination to the List page, you'll want to open the `Pages/List.php` file in the resource, not the resource class itself.

```
use Illuminate\Contracts\Pagination\CursorPaginator;
use Illuminate\Database\Eloquent\Builder;

protected function paginateTableQuery(Builder $query): CursorPaginator
{
    return $query->cursorPaginate(($this->getTableRecordsPerPage() === 'all') ? $query->count()
: $this->getTableRecordsPerPage());
}
```

Record URLs (clickable rows)

You may allow table rows to be completely clickable by using the `$table->recordUrl()` method:

```
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Model;

public function table(Table $table): Table
{
    return $table
        ->recordUrl(
            fn (Model $record): string => route('posts.edit', ['record' => $record]),
        );
}
```

In this example, clicking on each post will take you to the `posts.edit` route.

You may also open the URL in a new tab:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->openRecordUrlInNewTab();
}
```

If you'd like to override the URL for a specific column, or instead run an action when a column is clicked, see the [columns documentation](#).

Reordering records

To allow the user to reorder records using drag and drop in your table, you can use the `$table->reorderable()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->reorderable('sort');
}
```

If you're using mass assignment protection on your model, you will also need to add the `sort` attribute to the `$fillable` array there.

When making the table reorderable, a new button will be available on the table to toggle reordering.

Title	Slug	Author	Status
Customizing Filament's UI with a theme	theme-guide	Dennis Koch	⌚ reviewing
New Filament plugins in August	new-plugins-august	Adam Weston	⌚ published
Tips for building a great Filament plugin	plugin-tips	Zep Fietje	✍ draft
Top 5 best features of Filament	top-5-features	Ryan Chandler	⌚ reviewing
What is Filament?	what-is-filament	Dan Harrin	⌚ published

The `reorderable()` method accepts the name of a column to store the record order in. If you use something like `spatie/eloquent-sortable` with an order column such as `order_column`, you may use this instead:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->reorderable('order_column');
}
```

The `reorderable()` method also accepts a boolean condition as its second parameter, allowing you to conditionally enable reordering:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->reorderable('sort', auth()->user()->isAdmin());
}
```

Enabling pagination while reordering

Pagination will be disabled in reorder mode to allow you to move records between pages. It is generally bad UX to re-enable pagination while reordering, but if you are sure then you can use `$table->paginatedWhileReordering()`:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->paginatedWhileReordering();
}
```

Customizing the reordering trigger action

To customize the reordering trigger button, you may use the `reorderRecordsTriggerAction()` method, passing a closure that returns an action. All methods that are available to [customize action trigger buttons](#) can be used:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->reorderRecordsTriggerAction(
            fn (Action $action, bool $isReordering) => $action
                ->button()
                ->label($isReordering ? 'Disable reordering' : 'Enable reordering'),
        );
}
```

Drag and drop the records into order.			
Title	Slug	Author	Status
Customizing Filament's UI with a theme	theme-guide	Dennis Koch	⌚ reviewing
New Filament plugins in August	new-plugins-august	Adam Weston	🕒 published
Tips for building a great Filament plugin	plugin-tips	Zep Fietje	✍ draft
Top 5 best features of Filament	top-5-features	Ryan Chandler	⌚ reviewing
What is Filament?	what-is-filament	Dan Harrin	🕒 published

Customizing the table header

You can add a heading to a table using the `$table->heading()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->heading('Clients')
        ->columns([
            // ...
        ]);
}
```

You can also add a description below the heading using the `$table->description()` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->heading('Clients')
        ->description('Manage your clients here.')
        ->columns([
            // ...
        ]);
}
```

You can pass a view to the `[$table->header()]` method to customize the entire header:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->header(view('tables.header', [
            'heading' => 'Clients',
        ]))
        ->columns([
            // ...
        ]);
}
```

Polling table content

You may poll table content so that it refreshes at a set interval, using the `[$table->poll()]` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->poll('10s');
}
```

Deferring loading

Tables with lots of data might take a while to load, in which case you can load the table data asynchronously using the `[$table->deferLoading()]` method:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->deferLoading();
}
```

Searching records with Laravel Scout

While Filament doesn't provide a direct integration with [Laravel Scout](#), you may override methods to integrate it.

Use a `whereIn()` clause to filter the query for Scout results:

```
use App\Models\Post;
use Illuminate\Database\Eloquent\Builder;

protected function applySearchToTableQuery(Builder $query): Builder
{
    $this->applyColumnSearchesToTableQuery($query);

    if ($filled($search = $this->getTableSearch())) {
        $query->whereIn('id', Post::search($search)->keys());
    }

    return $query;
}
```

Scout uses this `whereIn()` method to retrieve results internally, so there is no performance penalty for using it.

The `applyColumnSearchesToTableQuery()` method ensures that searching individual columns will still work. You can replace that method with your own implementation if you want to use Scout for those search inputs as well.

For the global search input to show, at least one column in the table needs to be `searchable()`. Alternatively, if you are using Scout to control which columns are searchable already, you can simply pass `searchable()` to the entire table instead:

```
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->searchable();
}
```

Query string

Livewire ships with a feature to store data in the URL's query string, to access across requests.

With Filament, this allows you to store your table's filters, sort, search and pagination state in the URL.

To store the filters, sorting, and search state of your table in the query string:

```

use Livewire\Attributes\Url;

#[Url]
public bool $isTableReordering = false;

/**
 * @var array<string, mixed> | null
 */
#[Url]
public ?array $tableFilters = null;

#[Url]
public ?string $tableGrouping = null;

#[Url]
public ?string $tableGroupingDirection = null;

/**
 * @var ?string
 */
#[Url]
public $tableSearch = '';

#[Url]
public ?string $tableSortColumn = null;

#[Url]
public ?string $tableSortDirection = null;

```

Styling table rows

Striped table rows

To enable striped table rows, you can use the `striped()` method:

```

use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->striped();
}

```

The screenshot shows a table titled "Users" with the subtitle "Individuals who have registered for the application." A "New user" button is located in the top right corner. The table has columns for "Name", "Email address", and "Verified". Each row contains a checkbox, the user's name and email, a green checkmark or red X icon in the "Verified" column, and "Edit" and "Delete" buttons. At the bottom left, it says "Showing 1 to 5 of 51 results". On the right, there are buttons for "Per page" (set to 5), a page navigation bar (1, 2, 3, 4, ..., 10, 11, >), and a search bar.

<input type="checkbox"/>	Name	Email address	Verified	
<input type="checkbox"/>	Dan Harrin	dan@filamentphp.com	✓	<input checked="" type="button"/> Edit <input type="button"/> Delete
<input type="checkbox"/>	Ryan Chandler	ryan@filamentphp.com	✗	<input checked="" type="button"/> Edit <input type="button"/> Delete
<input type="checkbox"/>	Zep Fietje	zep@filamentphp.com	✗	<input checked="" type="button"/> Edit <input type="button"/> Delete
<input type="checkbox"/>	Dennis Koch	dennis@filamentphp.com	✓	<input checked="" type="button"/> Edit <input type="button"/> Delete
<input type="checkbox"/>	Adam Weston	adam@filamentphp.com	✓	<input checked="" type="button"/> Edit <input type="button"/> Delete

Custom row classes

You may want to conditionally style rows based on the record data. This can be achieved by specifying a string or array of CSS classes to be applied to the row using the `$table->recordClasses()` method:

```
use Closure;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Model;

public function table(Table $table): Table
{
    return $table
        ->recordClasses(fn (Model $record) => match ($record->status) {
            'draft' => 'opacity-30',
            'reviewing' => 'border-s-2 border-orange-600 dark:border-orange-300',
            'published' => 'border-s-2 border-green-600 dark:border-green-300',
            default => null,
        });
}
```

These classes are not automatically compiled by Tailwind CSS. If you want to apply Tailwind CSS classes that are not already used in Blade files, you should update your `content` configuration in `tailwind.config.js` to also scan for classes inside your directory: `'./app/Filament/**/* .php'`

Resetting the table

If you make changes to the table definition during a Livewire request, for example, when consuming a public property in the `table()` method, you may need to reset the table to ensure that the changes are applied. To do this, you can call the `resetTable()` method on the Livewire component:

```
$this->resetTable();
```

Global settings

To customize the default configuration that is used for all tables, you can call the static `configureUsing()` method from the `boot()` method of a service provider. The function will be run for each table that gets created:

```
use Filament\Tables\Enums\FiltersLayout;
use Filament\Tables\Table;

Table::configureUsing(function (Table $table): void {
    $table
        ->filtersLayout(FiltersLayout::AboveContentCollapsible)
        ->paginationPageOptions([10, 25, 50]);
});
```

Adding A Table To A Livewire Component

Setting up the Livewire component

First, generate a new Livewire component:

```
php artisan make:livewire ListProducts
```

Then, render your Livewire component on the page:

```
@livewire('list-products')
```

Alternatively, you can use a full-page Livewire component:

```
use App\Livewire>ListProducts;
use Illuminate\Support\Facades\Route;

Route::get('products', ListProducts::class);
```

Adding the table

There are 3 tasks when adding a table to a Livewire component class:

1. Implement the `HasTable` and `HasForms` interfaces, and use the `InteractsWithTable` and `InteractsWithForms` traits.
2. Add a `table()` method, which is where you configure the table. [Add the table's columns, filters, and actions.](#)
3. Make sure to define the base query that will be used to fetch rows in the table. For example, if you're listing products from your `Product` model, you will want to return `Product::query()`.

```

<?php

namespace App\Livewire;

use App\Models\Shop\Product;
use Filament\Forms\Concerns\InteractsWithForms;
use Filament\Forms\Contracts\HasForms;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Concerns\InteractsWithTable;
use Filament\Tables\Contracts\HasTable;
use Filament\Tables\Table;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class ListProducts extends Component implements HasForms, HasTable
{
    use InteractsWithTable;
    use InteractsWithForms;

    public function table(Table $table): Table
    {
        return $table
            ->query(Product::query())
            ->columns([
                TextColumn::make('name'),
            ])
            ->filters([
                // ...
            ])
            ->actions([
                // ...
            ])
            ->bulkActions([
                // ...
            ]);
    }

    public function render(): View
    {
        return view('livewire.list-products');
    }
}

```

Finally, in your Livewire component's view, render the table:

```

<div>
    {{ $this->table }}
</div>

```

Visit your Livewire component in the browser, and you should see the table.

Building a table for an Eloquent relationship

If you want to build a table for an Eloquent relationship, you can use the `relationship()` and `inverseRelationship()` methods on the `$table` instead of passing a `query()`. `HasMany`, `HasManyThrough`, `BelongsToMany`, `MorphMany` and `MorphToMany` relationships are compatible:

```
use App\Models\Category;
use Filament\Tables\Table;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

public Category $category;

public function table(Table $table): Table
{
    return $table
        ->relationship(fn (): BelongsToMany => $this->category->products())
        ->inverseRelationship('categories')
        ->columns([
            TextColumn::make('name'),
        ]);
}
```

In this example, we have a `$category` property which holds a `Category` model instance. The category has a relationship named `products`. We use a function to return the relationship instance. This is a many-to-many relationship, so the inverse relationship is called `categories`, and is defined on the `Product` model. We just need to pass the name of this relationship to the `inverseRelationship()` method, not the whole instance.

Now that the table is using a relationship instead of a plain Eloquent query, all actions will be performed on the relationship instead of the query. For example, if you use a `CreateAction`, the new product will be automatically attached to the category.

If your relationship uses a pivot table, you can use all pivot columns as if they were normal columns on your table, as long as they are listed in the `withPivot()` method of the relationship *and* inverse relationship definition.

Relationship tables are used in the Panel Builder as ["relation managers"](#). Most of the documented features for relation managers are also available for relationship tables. For instance, [attaching and detaching](#) and [associating and dissociating](#) actions.

Generating table Livewire components with the CLI

It's advised that you learn how to set up a Livewire component with the Table Builder manually, but once you are confident, you can use the CLI to generate a table for you.

```
php artisan make:livewire-table Products/ListProducts
```

This will ask you for the name of a prebuilt model, for example `Product`. Finally, it will generate a new `app/Livewire/Products>ListProducts.php` component, which you can customize.

Automatically generating table columns

Filament is also able to guess which table columns you want in the table, based on the model's database columns. You can use the `--generate` flag when generating your table:

```
php artisan make:livewire-table Products/ListProducts --generate
```

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Since the Table Builder works on Livewire components, you can use the [Livewire testing helpers](#). However, we have many custom testing helpers that you can use for tables:

Render

To ensure a table component renders, use the `assertSuccessful()` Livewire helper:

```
use function Pest\Livewire\livewire;

it('can render page', function () {
    livewire(ListPosts::class)->assertSuccessful();
});
```

To test which records are shown, you can use `assertCanSeeTableRecords()`, `assertCannotSeeTableRecords()` and `assertCountTableRecords()`:

```
use function Pest\Livewire\livewire;

it('cannot display trashed posts by default', function () {
    $posts = Post::factory()->count(4)->create();
    $trashedPosts = Post::factory()->trashed()->count(6)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts)
        ->assertCannotSeeTableRecords($trashedPosts)
        ->assertCountTableRecords(4);
});
```

If your table uses pagination, `assertCanSeeTableRecords()` will only check for records on the first page. To switch page, call `call('gotoPage', 2)`.

If your table uses `deferLoading()`, you should call `loadTable()` before `assertCanSeeTableRecords()`.

Columns

To ensure that a certain column is rendered, pass the column name to `assertCanRender TableColumn()`:

```
use function Pest\Livewire\livewire;

it('can render post titles', function () {
    Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanRenderTableColumn('title');
});
```

This helper will get the HTML for this column, and check that it is present in the table.

For testing that a column is not rendered, you can use `assertCannotRenderTableColumn()`:

```
use function Pest\Livewire\livewire;

it('can not render post comments', function () {
    Post::factory()->count(10)->create()

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCannotRenderTableColumn('comments');
});
```

This helper will assert that the HTML for this column is not shown by default in the present table.

Sorting

To sort table records, you can call `sortTable()`, passing the name of the column to sort by. You can use `'desc'` in the second parameter of `sortTable()` to reverse the sorting direction.

Once the table is sorted, you can ensure that the table records are rendered in order using

`assertCanSeeTableRecords()` with the `inOrder` parameter:

```
use function Pest\Livewire\livewire;

it('can sort posts by title', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->sortTable('title')
        ->assertCanSeeTableRecords($posts->sortBy('title'), inOrder: true)
        ->sortTable('title', 'desc')
        ->assertCanSeeTableRecords($posts->sortByDesc('title'), inOrder: true);
});
```

Searching

To search the table, call the `searchTable()` method with your search query.

You can then use `assertCanSeeTableRecords()` to check your filtered table records, and use `assertCannotSeeTableRecords()` to assert that some records are no longer in the table:

```
use function Pest\Livewire\livewire;

it('can search posts by title', function () {
    $posts = Post::factory()->count(10)->create();

    $title = $posts->first()->title;

    livewire(PostResource\Pages\ListPosts::class)
        ->searchTable($title)
        ->assertCanSeeTableRecords($posts->where('title', $title))
        ->assertCanNotSeeTableRecords($posts->where('title', '!=', $title));
});
```

To search individual columns, you can pass an array of searches to `searchTableColumns()`:

```
use function Pest\Livewire\livewire;

it('can search posts by title column', function () {
    $posts = Post::factory()->count(10)->create();

    $title = $posts->first()->title;

    livewire(PostResource\Pages\ListPosts::class)
        ->searchTableColumns(['title' => $title])
        ->assertCanSeeTableRecords($posts->where('title', $title))
        ->assertCanNotSeeTableRecords($posts->where('title', '!=', $title));
});
```

State

To assert that a certain column has a state or does not have a state for a record you can use

`assertTableColumnStateSet()` and `assertTableColumnStateNotSet()`:

```
use function Pest\Livewire\livewire;

it('can get post author names', function () {
    $posts = Post::factory()->count(10)->create();

    $post = $posts->first();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableColumnStateSet('author.name', $post->author->name, record: $post)
        ->assertTableColumnStateNotSet('author.name', 'Anonymous', record: $post);
});
```

To assert that a certain column has a formatted state or does not have a formatted state for a record you can use

`assertTableColumnFormattedStateSet()` and `assertTableColumnFormattedStateNotSet()`:

```
use function Pest\Livewire\livewire;

it('can get post author names', function () {
    $post = Post::factory(['name' => 'John Smith'])->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableColumnFormattedStateSet('author.name', 'Smith, John', record: $post)
        ->assertTableColumnFormattedStateNotSet('author.name', $post->author->name, record: $post);
});
```

Existence

To ensure that a column exists, you can use the `assertTableColumnExists()` method:

```
use function Pest\Livewire\livewire;

it('has an author column', function () {
    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableColumnExists('author');
});
```

You may pass a function as an additional argument in order to assert that a column passes a given "truth test". This is useful for asserting that a column has a specific configuration. You can also pass in a record as the third parameter, which is useful if your check is dependant on which table row is being rendered:

```
use function Pest\Livewire\livewire;
use Filament\Tables\Columns\TextColumn;

it('has an author column', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableColumnExists('author', function (TextColumn $column): bool {
            return $column->getDescriptionBelow() === $post->subtitle;
        }, $post);
});
```

Authorization

To ensure that a particular user cannot see a column, you can use the `assertTableColumnVisible()` and `assertTableColumnHidden()` methods:

```
use function Pest\Livewire\livewire;

it('shows the correct columns', function () {
    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableColumnVisible('created_at')
        ->assertTableColumnHidden('author');
});
```

Descriptions

To ensure a column has the correct description above or below you can use the `assertTableColumnHasDescription()` and `assertTableColumnDoesNotHaveDescription()` methods:

```
use function Pest\Livewire\livewire;

it('has the correct descriptions above and below author', function () {
    $post = Post::factory()->create();

    livewire(PostsTable::class)
        ->assertTableColumnHasDescription('author', 'Author! ↓↓', $post, 'above')
        ->assertTableColumnHasDescription('author', 'Author! ↑↑', $post)
        ->assertTableColumnDoesNotHaveDescription('author', 'Author! ↑↑', $post, 'above')
        ->assertTableColumnDoesNotHaveDescription('author', 'Author! ↓↓', $post);
});
```

Extra Attributes

To ensure that a column has the correct extra attributes, you can use the `assertTableColumnHasExtraAttributes()` and `assertTableColumnDoesNotHaveExtraAttributes()` methods:

```
use function Pest\Livewire\livewire;

it('displays author in red', function () {
    $post = Post::factory()->create();

    livewire(PostsTable::class)
        ->assertTableColumnHasExtraAttributes('author', ['class' => 'text-danger-500'], $post)
        ->assertTableColumnDoesNotHaveExtraAttributes('author', ['class' => 'text-primary-500'],
$post);
});
```

Select Columns

If you have a select column, you can ensure it has the correct options with `assertTableSelectColumnHasOptions()` and `assertTableSelectColumnDoesNotHaveOptions()`:

```
use function Pest\Livewire\livewire;

it('has the correct statuses', function () {
    $post = Post::factory()->create();

    livewire(PostsTable::class)
        ->assertTableSelectColumnHasOptions('status', ['unpublished' => 'Unpublished',
'published' => 'Published'], $post)
        ->assertTableSelectColumnDoesNotHaveOptions('status', ['archived' => 'Archived'],
$post);
});
```

Filters

To filter the table records, you can use the `filterTable()` method, along with `assertCanSeeTableRecords()` and `assertCannotSeeTableRecords()`:

```
use function Pest\Livewire\livewire;

it('can filter posts by `is_published`', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts)
        ->filterTable('is_published')
        ->assertCanSeeTableRecords($posts->where('is_published', true))
        ->assertCanNotSeeTableRecords($posts->where('is_published', false));
});
```

For a simple filter, this will just enable the filter.

If you'd like to set the value of a `SelectFilter` or `TernaryFilter`, pass the value as a second argument:

```
use function Pest\Livewire\livewire;

it('can filter posts by `author_id`', function () {
    $posts = Post::factory()->count(10)->create();

    $authorId = $posts->first()->author_id;

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts)
        ->filterTable('author_id', $authorId)
        ->assertCanSeeTableRecords($posts->where('author_id', $authorId))
        ->assertCanNotSeeTableRecords($posts->where('author_id', '!=', $authorId));
});
```

Resetting filters

To reset all filters to their original state, call `resetTableFilters()`:

```
use function Pest\Livewire\livewire;

it('can reset table filters', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->resetTableFilters();
});
```

Removing Filters

To remove a single filter you can use `removeTableFilter()`:

```
use function Pest\Livewire\livewire;

it('filters list by published', function () {
    $posts = Post::factory()->count(10)->create();

    $unpublishedPosts = $posts->where('is_published', false)->get();

    livewire(PostsTable::class)
        ->filterTable('is_published')
        ->assertCannotSeeTableRecords($unpublishedPosts)
        ->removeTableFilter('is_published')
        ->assertCanSeeTableRecords($posts);
});
```

To remove all filters you can use `removeTableFilters()`:

```
use function Pest\Livewire\livewire;

it('can remove all table filters', function () {
    $posts = Post::factory()->count(10)->forAuthor()->create();

    $unpublishedPosts = $posts
        ->where('is_published', false)
        ->where('author_id', $posts->first()->author->getKey());

    livewire(PostsTable::class)
        ->filterTable('is_published')
        ->filterTable('author', $author)
        ->assertCannotSeeTableRecords($unpublishedPosts)
        ->removeTableFilters()
        ->assertCanSeeTableRecords($posts);
});
```

Actions

Calling actions

You can call an action by passing its name or class to `callTableAction()`:

```
use function Pest\Livewire\livewire;

it('can delete posts', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableAction(DeleteAction::class, $post);

    $this->assertModelMissing($post);
});
```

This example assumes that you have a `DeleteAction` on your table. If you have a custom `Action::make('reorder')`, you may use `callTableAction('reorder')`.

For column actions, you may do the same, using `callTableColumnAction()`:

```
use function Pest\Livewire\livewire;

it('can copy posts', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableColumnAction('copy', $post);

    $this->assertDatabaseCount((new Post)->getTable(), 2);
});
```

For bulk actions, you may do the same, passing in multiple records to execute the bulk action against with `callTableBulkAction()`:

```
use function Pest\Livewire\livewire;

it('can bulk delete posts', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableBulkAction(DeleteBulkAction::class, $posts);

    foreach ($posts as $post) {
        $this->assertModelMissing($post);
    }
});
```

To pass an array of data into an action, use the `data` parameter:

```
use function Pest\Livewire\livewire;

it('can edit posts', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableAction(EditAction::class, $post, data: [
            'title' => $title = fake()->words(asText: true),
        ])
        ->assertHasNoTableActionErrors();

    expect($post->refresh())
        ->title->toBe($title);
});
```

Execution

To check if an action or bulk action has been halted, you can use `assertTableActionHalted()` / `assertTableBulkActionHalted()`:

```
use function Pest\Livewire\livewire;

it('will halt delete if post is flagged', function () {
    $posts= Post::factory()->count(2)->flagged()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableAction('delete', $posts->first())
        ->callTableBulkAction('delete', $posts)
        ->assertTableActionHalted('delete')
        ->assertTableBulkActionHalted('delete');

    $this->assertModelExists($post);
});
```

Errors

`assertHasNoTableActionErrors()` is used to assert that no validation errors occurred when submitting the action form.

To check if a validation error has occurred with the data, use `assertHasTableActionErrors()`, similar to `assertHasErrors()` in Livewire:

```
use function Pest\Livewire\livewire;

it('can validate edited post data', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->callTableAction(EditAction::class, $post, data: [
            'title' => null,
        ])
        ->assertHasTableActionErrors(['title' => ['required']]);
});
```

For bulk actions these methods are called `assertHasTableBulkActionErrors()` and `assertHasNoTableBulkActionErrors()`.

Pre-filled data

To check if an action or bulk action is pre-filled with data, you can use the `assertTableActionDataSet()` or `assertTableBulkActionDataSet()` method:

```
use function Pest\Livewire\livewire;

it('can load existing post data for editing', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->mountTableAction(EditAction::class, $post)
        ->assertTableActionDataSet([
            'title' => $post->title,
        ])
        ->setTableActionData([
            'title' => $title = fake()->words(asText: true),
        ])
        ->callMountedTableAction()
        ->assertHasNoTableActionErrors();

    expect($post->refresh())
        ->title->toBe($title);
});

```

Action state

To ensure that an action or bulk action exists or doesn't in a table, you can use the `assertTableActionExists()` / `assertTableActionDoesNotExist()` or `assertTableBulkActionExists()` / `assertTableBulkActionDoesNotExist()` method:

```
use function Pest\Livewire\livewire;

it('can publish but not unpublish posts', function () {
    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionExists('publish')
        ->assertTableActionDoesNotExist('unpublish')
        ->assertTableBulkActionExists('publish')
        ->assertTableBulkActionDoesNotExist('unpublish');
});

```

To ensure different sets of actions exist in the correct order, you can use the various "InOrder" assertions

```
use function Pest\Livewire\livewire;

it('has all actions in expected order', function () {
    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionsExistInOrder(['edit', 'delete'])
        ->assertTableBulkActionsExistInOrder(['restore', 'forceDelete'])
        ->assertTableHeaderActionsExistInOrder(['create', 'attach'])
        ->assertTableEmptyStateActionsExistInOrder(['create', 'toggle-trashed-filter']);
});

```

To ensure that an action or bulk action is enabled or disabled for a user, you can use the `assertTableActionEnabled()` / `assertTableActionDisabled()` or `assertTableBulkActionEnabled()` / `assertTableBulkActionDisabled()` methods:

```
use function Pest\Livewire\livewire;

it('can not publish, but can delete posts', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionDisabled('publish', $post)
        ->assertTableActionEnabled('delete', $post)
        ->assertTableBulkActionDisabled('publish')
        ->assertTableBulkActionEnabled('delete');

});
```

To ensure that an action or bulk action is visible or hidden for a user, you can use the `assertTableActionVisible()` / `assertTableActionHidden()` or `assertTableBulkActionVisible()` / `assertTableBulkActionHidden()` methods:

```
use function Pest\Livewire\livewire;

it('can not publish, but can delete posts', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionHidden('publish', $post)
        ->assertTableActionVisible('delete', $post)
        ->assertTableBulkActionHidden('publish')
        ->assertTableBulkActionVisible('delete');

});
```

Button Style

To ensure an action or bulk action has the correct label, you can use `assertTableActionHasLabel()` / `assertTableBulkActionHasLabel()` and `assertTableActionDoesNotHaveLabel()` / `assertTableBulkActionDoesNotHaveLabel()`:

```
use function Pest\Livewire\livewire;

it('delete actions have correct labels', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionHasLabel('delete', 'Archive Post')
        ->assertTableActionDoesNotHaveLabel('delete', 'Delete');
        ->assertTableBulkActionHasLabel('delete', 'Archive Post')
        ->assertTableBulkActionDoesNotHaveLabel('delete', 'Delete');

});
```

To ensure an action or bulk action's button is showing the correct icon, you can use `assertTableActionHasIcon()` / `assertTableBulkActionHasIcon()` or `assertTableActionDoesNotHaveIcon()` / `assertTableBulkActionDoesNotHaveIcon()`:

```
use function Pest\Livewire\livewire;

it('delete actions have correct icons', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionHasIcon('delete', 'heroicon-m-archive-box')
        ->assertTableActionDoesNotHaveIcon('delete', 'heroicon-m-trash');
        ->assertTableBulkActionHasIcon('delete', 'heroicon-m-archive-box')
        ->assertTableBulkActionDoesNotHaveIcon('delete', 'heroicon-m-trash');
});
});
```

To ensure that an action or bulk action's button is displaying the right color, you can use

`assertTableActionHasColor()` / `assertTableBulkActionHasColor()` or
`assertTableActionDoesNotHaveColor()` / `assertTableBulkActionDoesNotHaveColor()`:

```
use function Pest\Livewire\livewire;

it('delete actions have correct colors', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionHasColor('delete', 'warning')
        ->assertTableActionDoesNotHaveColor('delete', 'danger');
        ->assertTableBulkActionHasColor('delete', 'warning')
        ->assertTableBulkActionDoesNotHaveColor('delete', 'danger');
});
});
```

URL

To ensure an action or bulk action has the correct URL traits, you can use `assertTableActionHasUrl()`, `assertTableActionDoesNotHaveUrl()`, `assertTableActionShouldOpenUrlInNewTab()`, and `assertTableActionShouldNotOpenUrlInNewTab()`:

```
use function Pest\Livewire\livewire;

it('links to the correct Filament sites', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertTableActionHasUrl('filament', 'https://filamentphp.com/')
        ->assertTableActionDoesNotHaveUrl('filament', 'https://github.com/filamentphp/filament')
        ->assertTableActionShouldOpenUrlInNewTab('filament')
        ->assertTableActionShouldNotOpenUrlInNewTab('github');
});
});
```

Summaries

To test that a summary calculation is working, you may use the `assertTableColumnSummarySet()` method:

```
use function Pest\Livewire\livewire;

it('can average values in a column', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts)
        ->assertTableColumnSummarySet('rating', 'average', $posts->avg('rating'));
});

});
```

The first argument is the column name, the second is the summarizer ID, and the third is the expected value.

You may set a summarizer ID by passing it to the `make()` method:

```
use Filament\Tables\Columns\Summarizers\Average;
use Filament\Tables\Columns\TextColumn;

TextColumn::make('rating')
    ->summarize(Average::make('average'))
```

The ID should be unique between summarizers in that column.

Summarizing only one pagination page

To calculate the average for only one pagination page, use the `isCurrentPaginationPageOnly` argument:

```
use function Pest\Livewire\livewire;

it('can average values in a column', function () {
    $posts = Post::factory()->count(20)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts->take(10))
        ->assertTableColumnSummarySet('rating', 'average', $posts->take(10)->avg('rating'),
isCurrentPaginationPageOnly: true);
});
```

Testing a range summarizer

To test a range, pass the minimum and maximum value into a tuple-style `[$minimum, $maximum]` array:

```
use function Pest\Livewire\livewire;

it('can average values in a column', function () {
    $posts = Post::factory()->count(10)->create();

    livewire(PostResource\Pages\ListPosts::class)
        ->assertCanSeeTableRecords($posts)
        ->assertTableColumnSummarySet('rating', 'range', [$posts->min('rating'), $posts-
>max('rating')]);
});
```

Upgrade Guide

If you see anything missing from this guide, please do not hesitate to [make a pull request](#) to our repository! Any help is appreciated!

New requirements

- Laravel v10.0+
- Livewire v3.0+

Please upgrade Filament before upgrading to Livewire v3. Instructions on how to upgrade Livewire can be found [here](#).

Upgrading automatically

The easiest way to upgrade your app is to run the automated upgrade script. This script will automatically upgrade your application to the latest version of Filament, and make changes to your code which handle most breaking changes.

```
composer require filament/upgrade:"^3.2" -W --dev
vendor/bin/filament-v3
```

Make sure to carefully follow the instructions, and review the changes made by the script. You may need to make some manual changes to your code afterwards, but the script should handle most of the repetitive work for you.

Finally, you must run `php artisan filament:install` to finalize the Filament v3 installation. This command must be run for all new Filament projects.

You can now `composer remove filament/upgrade` as you don't need it anymore.

Some plugins you're using may not be available in v3 just yet. You could temporarily remove them from your `composer.json` file until they've been upgraded, replace them with a similar plugins that are v3-compatible, wait for the plugins to be upgraded before upgrading your app, or even write PRs to help the authors upgrade them.

Upgrading manually

After upgrading the dependency via Composer, you should execute `php artisan filament:upgrade` in order to clear any Laravel caches and publish the new frontend assets.

High-impact changes

Config file renamed and combined with other Filament packages

Only one config file is now used for all Filament packages. Most configuration has been moved into other parts of the codebase, and little remains. You should use the v3 documentation as a reference when replace the configuration options you did modify. To publish the new configuration file and remove the old one, run:

```
php artisan vendor:publish --tag=filament-config --force
rm config/tables.php
```

`TABLES_FILESYSTEM_DRIVER` .env variable

The `TABLES_FILESYSTEM_DRIVER` .env variable has been renamed to `FILAMENT_FILESYSTEM_DISK`. This is to make it more consistent with Laravel, as Laravel v9 introduced this change as well. Please ensure that you update your .env files accordingly, and don't forget production!

New `@filamentScripts` and `@filamentStyles` Blade directives

The `@filamentScripts` and `@filamentStyles` Blade directives must be added to your Blade layout file/s. Since Livewire v3 no longer uses similar directives, you can replace `@livewireScripts` with `@filamentScripts` and `@livewireStyles` with `@filamentStyles`.

CSS file removed

The CSS file for form components, `module.esm.css`, has been removed. Check `resources/css/app.css`. That CSS is now automatically loaded by `@filamentStyles`.

JavaScript files removed

You no longer need to import the `FormsAlpinePlugin` in your JavaScript files. Alpine plugins are now automatically loaded by `@filamentScripts`.

Heroicons have been updated to v2

The Heroicons library has been updated to v2. This means that any icons you use in your app may have changed names. You can find a list of changes [here](#).

Medium-impact changes**Secondary color**

Filament v2 had a `secondary` color for many components which was gray. All references to `secondary` should be replaced with `gray` to preserve the same appearance. This frees `secondary` to be registered to a new custom color of your choice.

`BadgeColumn::enum()` removed

You can use a `formatStateUsing()` function to transform text.

Enum classes moved

The following enum classes have moved:

- `Filament\Tables\Actions\Position` has moved to `Filament\Tables\Enums\ActionsPosition`.
- `Filament\Tables\Actions\RecordCheckboxPosition` has moved to `Filament\Tables\Enums\RecordCheckboxPosition`.
- `Filament\Tables\Filters\Layout` has moved to `Filament\Tables\Enums\FiltersLayout`.

Chapter 4

Notifications

Installation

The Notifications package is pre-installed with the [Panel Builder](#). This guide is for using the Notifications package in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Require the Notifications package using Composer:

```
composer require filament/notifications:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --notifications  
npm install  
npm run dev
```

Existing Laravel projects

Run the following command to install the Notifications package assets:

```
php artisan filament:install --notifications
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting  
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament `preset` (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()>getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @livewire('notifications')

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Upgrading from Filament v2? Please review the [upgrade guide](#).

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post-autoload-dump` hook, you can run the command yourself:

```
composer update

php artisan filament:upgrade
```

Sending Notifications

Overview

To start, make sure the package is [installed](#) - `@livewire('notifications')` should be in your Blade layout somewhere.

Notifications are sent using a `Notification` object that's constructed through a fluent API. Calling the `send()` method on the `Notification` object will dispatch the notification and display it in your application. As the session is used to flash notifications, they can be sent from anywhere in your code, including JavaScript, not just Livewire components.

```
<?php

namespace App\Livewire;

use Filament\Notifications\Notification;
use Livewire\Component;

class EditPost extends Component
{
    public function save(): void
    {
        // ...

        Notification::make()
            ->title('Saved successfully')
            ->success()
            ->send();
    }
}
```



Saved



Setting a title

The main message of the notification is shown in the title. You can set the title as follows:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->send();
```

The title text can contain basic, safe HTML elements. To generate safe HTML with Markdown, you can use the

```
Str::markdown() helper: title(Str::markdown('Saved **successfully**'))
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .send()
```

Setting an icon

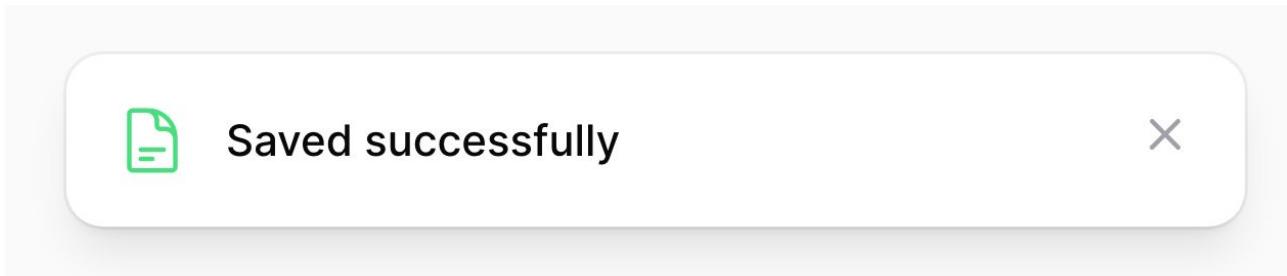
Optionally, a notification can have an icon that's displayed in front of its content. You may also set a color for the icon, which is gray by default:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->icon('heroicon-o-document-text')
    ->iconColor('success')
    ->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .icon('heroicon-o-document-text')
    .iconColor('success')
    .send()
```



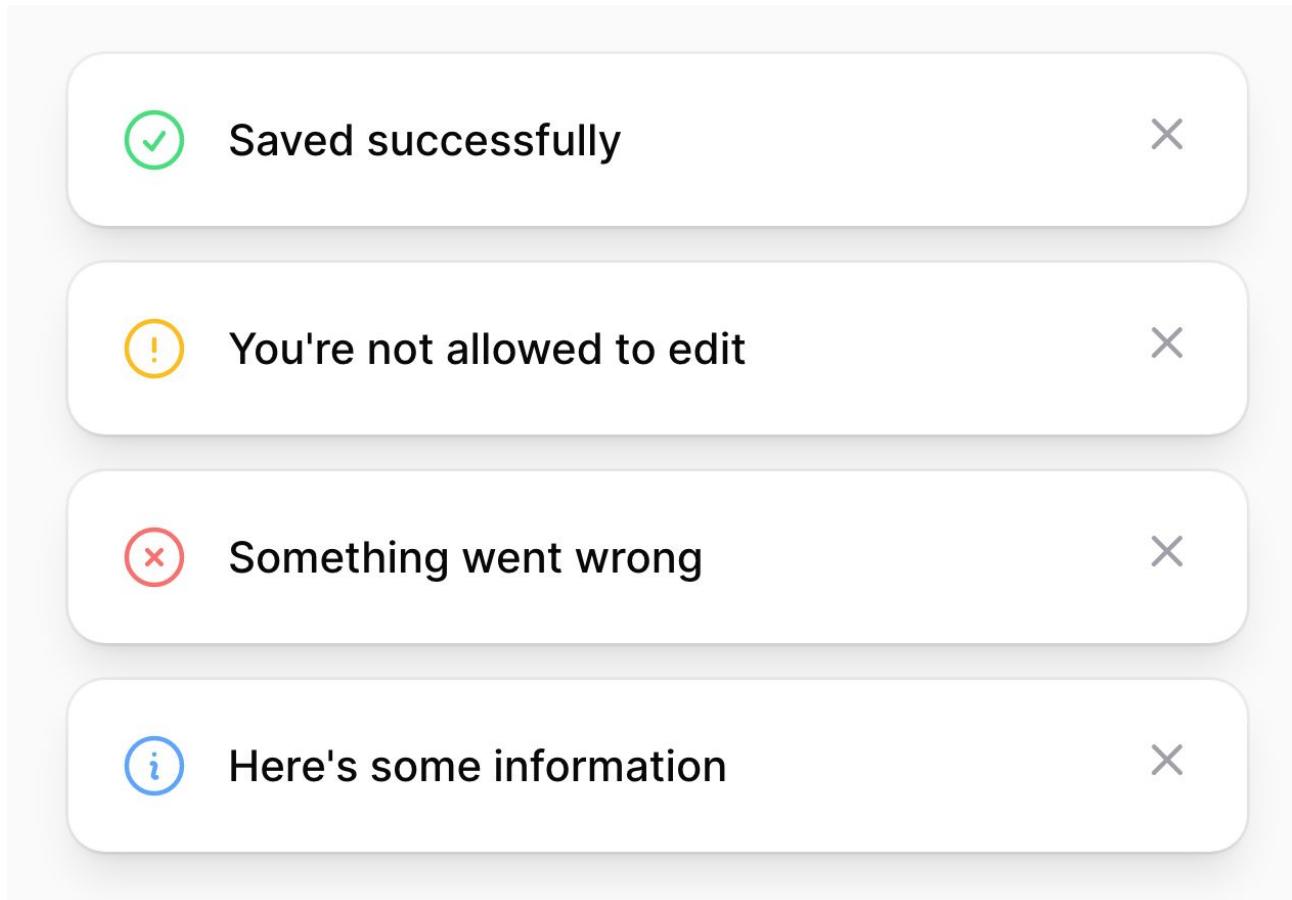
Notifications often have a status like `success`, `warning`, `danger` or `info`. Instead of manually setting the corresponding icons and colors, there's a `status()` method which you can pass the status. You may also use the dedicated `success()`, `warning()`, `danger()` and `info()` methods instead. So, cleaning up the above example would look like this:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .send()
```



Setting a background color

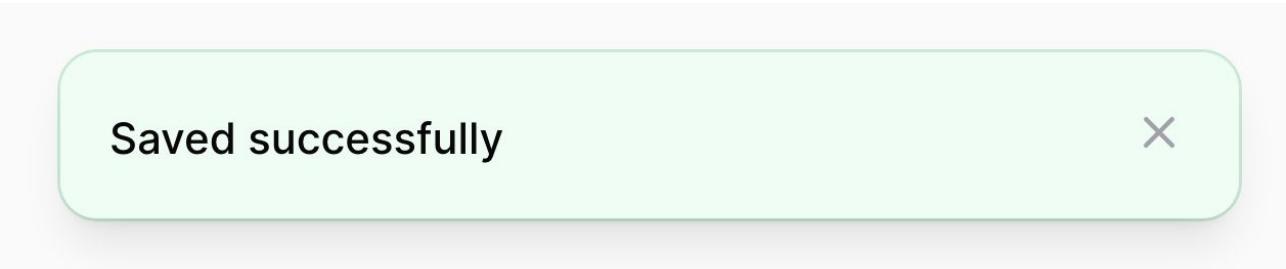
Notifications have no background color by default. You may want to provide additional context to your notification by setting a color as follows:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->color('success') // [tl! focus]
    ->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .color('success') // [tl! focus]
    .send()
```



Saved successfully



Setting a duration

By default, notifications are shown for 6 seconds before they're automatically closed. You may specify a custom duration value in milliseconds as follows:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->duration(5000)
    ->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .duration(5000)
    .send()
```

If you prefer setting a duration in seconds instead of milliseconds, you can do so:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->seconds(5)
    ->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .seconds(5)
    .send()
```

You might want some notifications to not automatically close and require the user to close them manually. This can be achieved by making the notification persistent:

```
use Filament\Notifications\Notification;

Notification::make()
->title('Saved successfully')
->success()
->persistent()
->send();
```

Or with JavaScript:

```
new FilamentNotification()
.title('Saved successfully')
.success()
.persistent()
.send()
```

Setting body text

Additional notification text can be shown in the `body()`:

```
use Filament\Notifications\Notification;

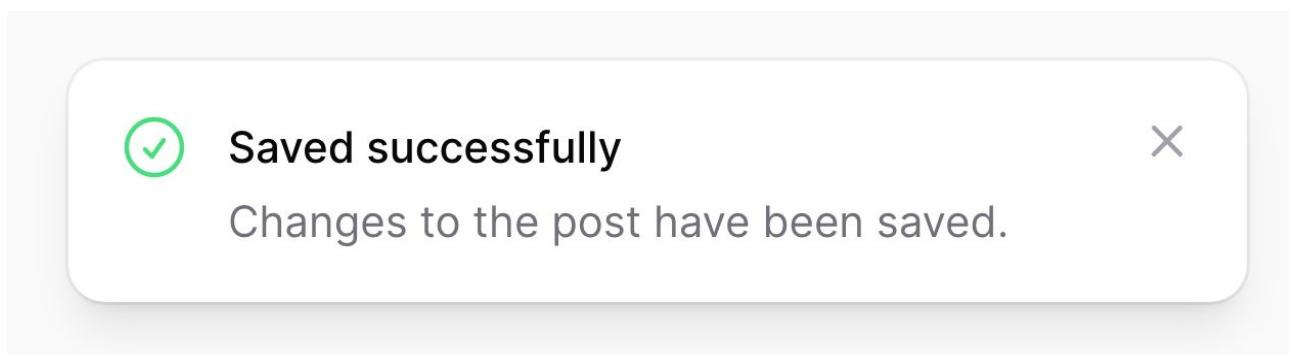
Notification::make()
->title('Saved successfully')
->success()
->body('Changes to the post have been saved.')
->send();
```

The body text can contain basic, safe HTML elements. To generate safe HTML with Markdown, you can use the

`Str::markdown()` helper: `body(Str::markdown('Changes to the **post** have been saved.'))`

Or with JavaScript:

```
new FilamentNotification()
.title('Saved successfully')
.success()
.body('Changes to the post have been saved.')
.send()
```



Adding actions to notifications

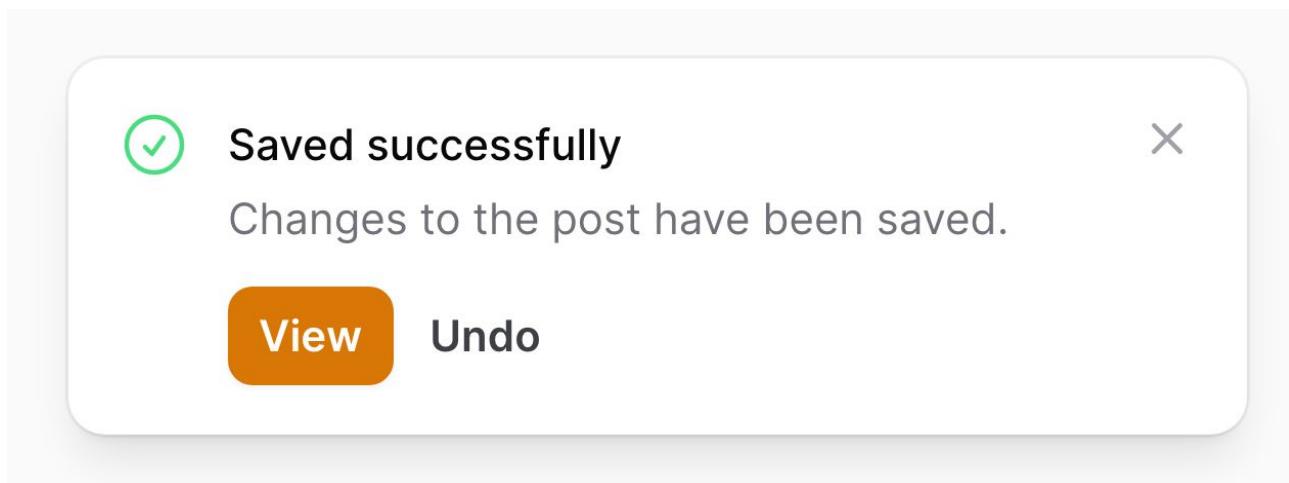
Notifications support [Actions](#), which are buttons that render below the content of the notification. They can open a URL or dispatch a Livewire event. Actions can be defined as follows:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->body('Changes to the post have been saved.')
    ->actions([
        Action::make('view')
            ->button(),
        Action::make('undo')
            ->color('gray'),
    ])
->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .body('Changes to the post have been saved.')
    .actions([
        new FilamentNotificationAction('view')
            .button(),
        new FilamentNotificationAction('undo')
            .color('gray'),
    ])
    .send()
```



You can learn more about how to style action buttons [here](#).

Opening URLs from notification actions

You can open a URL, optionally in a new tab, when clicking on an action:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->body('Changes to the post have been saved.')
    ->actions([
        Action::make('view')
            ->button()
            ->url(route('posts.show', $post), shouldOpenInNewTab: true)
        Action::make('undo')
            ->color('gray'),
    ])
->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .body('Changes to the post have been saved.')
    .actions([
        new FilamentNotificationAction('view')
            .button()
            .url('/view')
            .openUrlInNewTab(),
        new FilamentNotificationAction('undo')
            .color('gray'),
    ])
    .send()
```

Dispatching Livewire events from notification actions

Sometimes you want to execute additional code when a notification action is clicked. This can be achieved by setting a Livewire event which should be dispatched on clicking the action. You may optionally pass an array of data, which will be available as parameters in the event listener on your Livewire component:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->body('Changes to the post have been saved.')
    ->actions([
        Action::make('view')
            ->button()
            ->url(route('posts.show', $post), shouldOpenInNewTab: true),
        Action::make('undo')
            ->color('gray')
            ->dispatch('undoEditingPost', [$post->id]),
    ])
->send();
```

You can also `dispatchSelf` and `dispatchTo`:

```
Action::make('undo')
->color('gray')
->dispatchSelf('undoEditingPost', [$post->id])

Action::make('undo')
->color('gray')
->dispatchTo('another_component', 'undoEditingPost', [$post->id])
```

Or with JavaScript:

```
new FilamentNotification()
.title('Saved successfully')
.success()
.body('Changes to the post have been saved.')
.actions([
    new FilamentNotificationAction('view')
        .button()
        .url('/view')
        .openUrlInNewTab(),
    new FilamentNotificationAction('undo')
        .color('gray')
        .dispatch('undoEditingPost'),
])
.send()
```

Similarly, `dispatchSelf` and `dispatchTo` are also available:

```
new FilamentNotificationAction('undo')
.color('gray')
.dispatchSelf('undoEditingPost')

new FilamentNotificationAction('undo')
.color('gray')
.dispatchTo('another_component', 'undoEditingPost')
```

Closing notifications from actions

After opening a URL or dispatching an event from your action, you may want to close the notification right away:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->body('Changes to the post have been saved.')
    ->actions([
        Action::make('view')
            ->button()
            ->url(route('posts.show', $post), shouldOpenInNewTab: true),
        Action::make('undo')
            ->color('gray')
            ->dispatch('undoEditingPost', [$post->id])
            ->close(),
    ])
->send();
```

Or with JavaScript:

```
new FilamentNotification()
    .title('Saved successfully')
    .success()
    .body('Changes to the post have been saved.')
    .actions([
        new FilamentNotificationAction('view')
            .button()
            .url('/view')
            .openUrlInNewTab(),
        new FilamentNotificationAction('undo')
            .color('gray')
            .dispatch('undoEditingPost')
            .close(),
    ])
    .send()
```

Using the JavaScript objects

The JavaScript objects (`FilamentNotification` and `FilamentNotificationAction`) are assigned to `window.FilamentNotification` and `window.FilamentNotificationAction`, so they are available in on-page scripts.

You may also import them in a bundled JavaScript file:

```
import { Notification, NotificationAction } from
'../../vendor/filament/notifications/dist/index.js'

// ...
```

Closing a notification with JavaScript

Once a notification has been sent, you can close it on demand by dispatching a browser event on the window called `close-notification`.

The event needs to contain the ID of the notification you sent. To get the ID, you can use the `getId()` method on the `Notification` object:

```
use Filament\Notifications\Notification;

$notification = Notification::make()
    ->title('Hello')
    ->persistent()
    ->send()

$notificationId = $notification->getId()
```

To close the notification, you can dispatch the event from Livewire:

```
$this->dispatch('close-notification', id: $notificationId);
```

Or from JavaScript, in this case Alpine.js:

```
<button x-on:click="$dispatch('close-notification', { id: notificationId })" type="button">
    Close Notification
</button>
```

If you are able to retrieve the notification ID, persist it, and then use it to close the notification, that is the recommended approach, as IDs are generated uniquely, and you will not risk closing the wrong notification. However, if it is not possible to persist the random ID, you can pass in a custom ID when sending the notification:

```
use Filament\Notifications\Notification;

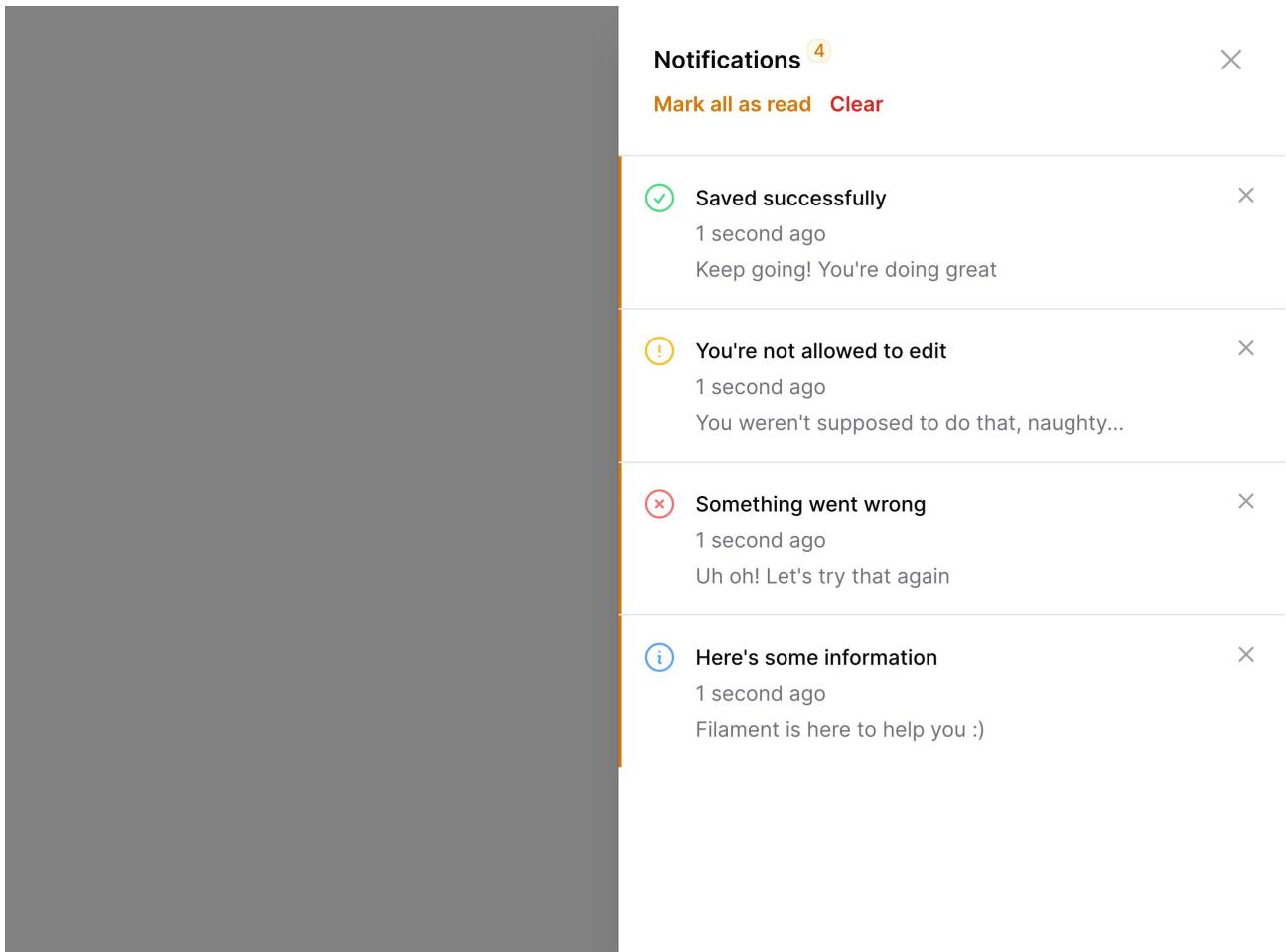
Notification::make('greeting')
    ->title('Hello')
    ->persistent()
    ->send()
```

In this case, you can close the notification by dispatching the event with the custom ID:

```
<button x-on:click="$dispatch('close-notification', { id: 'greeting' })" type="button">
    Close Notification
</button>
```

Please be aware that if you send multiple notifications with the same ID, you may experience unexpected side effects, so random IDs are recommended.

Database Notifications



Setting up the notifications database table

Before we start, make sure that the [Laravel notifications table](#) is added to your database:

```
# Laravel 11 and higher
php artisan make:notifications-table

# Laravel 10
php artisan notifications:table
```

If you're using PostgreSQL, make sure that the `[data]` column in the migration is using `json()`: `$table->json('data')`.

If you're using UUIDs for your `User` model, make sure that your `[notifiable]` column is using `uuidMorphs()`: `$table->uuidMorphs('notifiable')`.

Rendering the database notifications modal

If you want to add database notifications to a panel, [follow this part of the guide](#).

If you'd like to render the database notifications modal outside of the [Panel Builder](#), you'll need to add a new Livewire component to your Blade layout:

```
@livewire('database-notifications')
```

To open the modal, you must have a "trigger" button in your view. Create a new trigger button component in your app, for instance at `/resources/views/filament/notifications/database-notifications-trigger.blade.php`:

```
<button type="button">
    Notifications {{ $unreadNotificationsCount }} unread
</button>
```

`$unreadNotificationsCount` is a variable automatically passed to this view, which provides it with a real-time count of unread notifications the user has.

In the service provider, point to this new trigger view:

```
use Filament\Notifications\Livewire\DatabaseNotifications;

DatabaseNotifications::trigger('filament.notifications.database-notifications-trigger');
```

Now, click on the trigger button that is rendered in your view. A modal should appear containing your database notifications when clicked!

Adding the database notifications modal to a panel

You can enable database notifications in a panel's [configuration](#):

```
use Filament\Panel;

public function panel(Panel $panel): Panel
{
    return $panel
        // ...
        ->databaseNotifications();
}
```

To learn more, visit the [Panel Builder documentation](#).

Sending database notifications

There are several ways to send database notifications, depending on which one suits you best.

You may use our fluent API:

```
use Filament\Notifications\Notification;

$recipient = auth()->user();

Notification::make()
    ->title('Saved successfully')
    ->sendToDatabase($recipient);
```

Or, use the `notify()` method:

```
use Filament\Notifications\Notification;

$recipient = auth()->user();

$recipient->notify(
    Notification::make()
        ->title('Saved successfully')
        ->toDatabase(),
);

```

Laravel sends database notifications using the queue. Ensure your queue is running in order to receive the notifications.

Alternatively, use a traditional [Laravel notification class](#) by returning the notification from the `toDatabase()` method:

```
use App\Models\User;
use Filament\Notifications\Notification;

public function toDatabase(User $notifiable): array
{
    return Notification::make()
        ->title('Saved successfully')
        ->getDatabaseMessage();
}
```

Receiving database notifications

Without any setup, new database notifications will only be received when the page is first loaded.

Polling for new database notifications

Polling is the practice of periodically making a request to the server to check for new notifications. This is a good approach as the setup is simple, but some may say that it is not a scalable solution as it increases server load.

By default, Livewire polls for new notifications every 30 seconds:

```
use Filament\Notifications\Livewire\DatabaseNotifications;

DatabaseNotifications::pollingInterval('30s');
```

You may completely disable polling if you wish:

```
use Filament\Notifications\Livewire\DatabaseNotifications;

DatabaseNotifications::pollingInterval(null);
```

Using Echo to receive new database notifications with websockets

Alternatively, the package has a native integration with [Laravel Echo](#). Make sure Echo is installed, as well as a [server-side websockets integration](#) like Pusher.

Once websockets are set up, after sending a database notification you may dispatch a `DatabaseNotificationsSent` event, which will immediately fetch new notifications for that user:

```
use Filament\Notifications\Events\DatabaseNotificationsSent;
use Filament\Notifications\Notification;

$recipient = auth()->user();

Notification::make()
->title('Saved successfully')
->sendToDatabase($recipient);

event(new DatabaseNotificationsSent($recipient));
```

Marking database notifications as read

There is a button at the top of the modal to mark all notifications as read at once. You may also add [Actions](#) to notifications, which you can use to mark individual notifications as read. To do this, use the `markAsRead()` method on the action:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
->title('Saved successfully')
->success()
->body('Changes to the post have been saved.')
->actions([
    Action::make('view')
        ->button()
        ->markAsRead(),
])
->send();
```

Alternatively, you may use the `markAsUnread()` method to mark a notification as unread:

```
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

Notification::make()
->title('Saved successfully')
->success()
->body('Changes to the post have been saved.')
->actions([
    Action::make('markAsUnread')
        ->button()
        ->markAsUnread(),
])
->send();
```

Opening the database notifications modal

Instead of rendering the trigger button as described above, you can always open the database notifications modal from anywhere by dispatching an `open-modal` browser event:

```
<button  
  x-data="{}"  
  x-on:click="$dispatch('open-modal', { id: 'database-notifications' })"  
  type="button"  
>  
  Notifications  
</button>
```

Broadcast Notifications

Overview

To start, make sure the package is [installed](#) - `@livewire('notifications')` should be in your Blade layout somewhere.

By default, Filament will send flash notifications via the Laravel session. However, you may wish that your notifications are "broadcast" to a user in real-time, instead. This could be used to send a temporary success notification from a queued job after it has finished processing.

We have a native integration with [Laravel Echo](#). Make sure Echo is installed, as well as a [server-side websockets integration](#) like Pusher.

Sending broadcast notifications

There are several ways to send broadcast notifications, depending on which one suits you best.

You may use our fluent API:

```
use Filament\Notifications\Notification;

$recipient = auth()->user();

Notification::make()
    ->title('Saved successfully')
    ->broadcast($recipient);
```

Or, use the `notify()` method:

```
use Filament\Notifications\Notification;

$recipient = auth()->user();

$recipient->notify(
    Notification::make()
        ->title('Saved successfully')
        ->toBroadcast(),
)
```

Alternatively, use a traditional [Laravel notification class](#) by returning the notification from the `toBroadcast()` method:

```
use App\Models\User;
use Filament\Notifications\Notification;
use Illuminate\Notifications\Messages\BroadcastMessage;

public function toBroadcast(User $notifiable): BroadcastMessage
{
    return Notification::make()
        ->title('Saved successfully')
        ->getBroadcastMessage();
}
```

Customizing Notifications

Overview

Notifications come fully styled out of the box. However, if you want to apply your own styling or use a custom view to render notifications, there are multiple options.

Styling notifications

Notifications have dedicated CSS classes you can hook into to apply your own styling. Open the inspector in your browser to find out which classes you need to target.

Positioning notifications

You can configure the alignment of the notifications in a service provider or middleware, by calling

`Notifications::alignment()` and `Notifications::verticalAlignment()`. You can pass `Alignment::Start`, `Alignment::Center`, `Alignment::End`, `VerticalAlignment::Start`, `VerticalAlignment::Center` or `VerticalAlignment::End`:

```
use Filament\Notifications\Livewire\Notifications;
use Filament\Support\Enums\Alignment;
use Filament\Support\Enums\VerticalAlignment;

Notifications::alignment(Alignment::Start);
Notifications::verticalAlignment(VerticalAlignment::End);
```

Using a custom notification view

If your desired customization can't be achieved using the CSS classes above, you can create a custom view to render the notification. To configure the notification view, call the static `configureUsing()` method inside a service provider's `boot()` method and specify the view to use:

```
use Filament\Notifications\Notification;

Notification::configureUsing(function (Notification $notification): void {
    $notification->view('filament-notifications.notification');
});
```

Next, create the view, in this example `resources/views/notifications/notification.blade.php`. The view should use the package's base notification component for the notification functionality and pass the available `$notification` variable through the `notification` attribute. This is the bare minimum required to create your own notification view:

```
<x-filament-notifications::notification :notification="$notification">
{{-- Notification content --}}
</x-filament-notifications::notification>
```

Getters for all notification properties will be available in the view. So, a custom notification view might look like this:

```
<x-filament-notifications::notification
:notification="$notification"
class="flex w-80 rounded-lg transition duration-200"
x-transition:enter-start="opacity-0"
x-transition:leave-end="opacity-0"
>
<h4>
    {{ $getTitle() }}
</h4>

<p>
    {{ $getDate() }}
</p>

<p>
    {{ $getBody() }}
</p>

<span x-on:click="close">
    Close
</span>
</x-filament-notifications::notification>
```

Using a custom notification object

Maybe your notifications require additional functionality that's not defined in the package's `Notification` class. Then you can create your own `Notification` class, which extends the package's `Notification` class. For example, your notification design might need a size property.

Your custom `Notification` class in `app/Notifications/Notification.php` might contain:

```
<?php

namespace App\Notifications;

use Filament\Notifications\Notification as BaseNotification;

class Notification extends BaseNotification
{
    protected string $size = 'md';

    public function toArray(): array
    {
        return [
            ...parent::toArray(),
            'size' => $this->getSize(),
        ];
    }

    public static function fromArray(array $data): static
    {
        return parent::fromArray($data)->size($data['size']);
    }

    public function size(string $size): static
    {
        $this->size = $size;

        return $this;
    }

    public function getSize(): string
    {
        return $this->size;
    }
}
```

Next, you should bind your custom `Notification` class into the container inside a service provider's `register()` method:

```
use App\Notifications\Notification;
use Filament\Notifications\Notification as BaseNotification;

$this->app->bind(BaseNotification::class, Notification::class);
```

You can now use your custom `Notification` class in the same way as you would with the default `Notification` object.

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Testing session notifications

To check if a notification was sent using the session, use the `assertNotified()` helper:

```
use function Pest\LiveWire\livewire;

it('sends a notification', function () {
    livewire(CreatePost::class)
        ->assertNotified();
});
```

```
use Filament\Notifications\Notification;

it('sends a notification', function () {
    Notification::assertNotified();
});
```

```
use function Filament\Notifications\Testing\assertNotified;

it('sends a notification', function () {
    assertNotified();
});
```

You may optionally pass a notification title to test for:

```
use Filament\Notifications\Notification;
use function Pest\LiveWire\livewire;

it('sends a notification', function () {
    livewire(CreatePost::class)
        ->assertNotified('Unable to create post');
});
```

Or test if the exact notification was sent:

```
use Filament\Notifications\Notification;
use function Pest\Livewire\livewire;

it('sends a notification', function () {
    livewire(CreatePost::class)
        ->assertNotified(
            Notification::make()
                ->danger()
                ->title('Unable to create post')
                ->body('Something went wrong.'),
        );
});
```

Conversely, you can assert that a notification was not sent:

```
use Filament\Notifications\Notification;
use function Pest\Livewire\livewire;

it('does not send a notification', function () {
    livewire(CreatePost::class)
        ->assertNotNotified()
        // or
        ->assertNotNotified('Unable to create post')
        // or
        ->assertNotified(
            Notification::make()
                ->danger()
                ->title('Unable to create post')
                ->body('Something went wrong.'),
        );
});
```

Upgrade Guide

If you see anything missing from this guide, please do not hesitate to [make a pull request](#) to our repository! Any help is appreciated!

New requirements

- Laravel v10.0+
- Livewire v3.0+

Please upgrade Filament before upgrading to Livewire v3. Instructions on how to upgrade Livewire can be found [here](#).

Upgrading automatically

The easiest way to upgrade your app is to run the automated upgrade script. This script will automatically upgrade your application to the latest version of Filament, and make changes to your code which handle most breaking changes.

```
composer require filament/upgrade:"^3.2" -W --dev
vendor/bin/filament-v3
```

Make sure to carefully follow the instructions, and review the changes made by the script. You may need to make some manual changes to your code afterwards, but the script should handle most of the repetitive work for you.

Finally, you must run `php artisan filament:install` to finalize the Filament v3 installation. This command must be run for all new Filament projects.

You can now `composer remove filament/upgrade` as you don't need it anymore.

Some plugins you're using may not be available in v3 just yet. You could temporarily remove them from your `composer.json` file until they've been upgraded, replace them with a similar plugins that are v3-compatible, wait for the plugins to be upgraded before upgrading your app, or even write PRs to help the authors upgrade them.

Upgrading manually

After upgrading the dependency via Composer, you should execute `php artisan filament:upgrade` in order to clear any Laravel caches and publish the new frontend assets.

High-impact changes

Config file renamed and combined with other Filament packages

Only one config file is now used for all Filament packages. Most configuration has been moved into other parts of the codebase, and little remains. You should use the v3 documentation as a reference when replace the configuration options you did modify. To publish the new configuration file and remove the old one, run:

```
php artisan vendor:publish --tag=filament-config --force
rm config/notifications.php
```

New `@filamentScripts` and `@filamentStyles` Blade directives

The `@filamentScripts` and `@filamentStyles` Blade directives must be added to your Blade layout file/s. Since Livewire v3 no longer uses similar directives, you can replace `@livewireScripts` with `@filamentScripts` and `@livewireStyles` with `@filamentStyles`.

JavaScript assets

You no longer need to import the `NotificationsAlpinePlugin` in your JavaScript files. Alpine plugins are now automatically loaded by `@filamentScripts`.

Heroicons have been updated to v2

The Heroicons library has been updated to v2. This means that any icons you use in your app may have changed names. You can find a list of changes [here](#).

Medium-impact changes

Secondary color

Filament v2 had a `secondary` color for many components which was gray. All references to `secondary` should be replaced with `gray` to preserve the same appearance. This frees `secondary` to be registered to a new custom color of your choice.

Notification JS objects

The `Notification` JavaScript object has been renamed to `FilamentNotification` to avoid conflicts with the native browser `Notification` object. The same has been done for `NotificationAction` (now `FilamentNotificationAction`) and `NotificationActionGroup` (now `FilamentNotificationActionGroup`) for consistency.

Chapter 5

Actions

Installation

The Actions package is pre-installed with the [Panel Builder](#). This guide is for using the Actions package in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

Require the Actions package using Composer:

```
composer require filament/actions:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --actions
npm install
npm run dev
```

Existing Laravel projects

Run the following command to install the Actions package assets:

```
php artisan filament:install --actions
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament [preset](#) (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post-autoload-dump` hook, you can run the command yourself:

```
composer update
```

```
php artisan filament:upgrade
```

Overview

What is an action?

"Action" is a word that is used quite a bit within the Laravel community. Traditionally, action PHP classes handle "doing" something in your application's business logic. For instance, logging a user in, sending an email, or creating a new user record in the database.

In Filament, actions also handle "doing" something in your app. However, they are a bit different from traditional actions. They are designed to be used in the context of a user interface. For instance, you might have a button to delete a client record, which opens a modal to confirm your decision. When the user clicks the "Delete" button in the modal, the client is deleted. This whole workflow is an "action".

```
Action::make('delete')
->requiresConfirmation()
->action(fn () => $this->client->delete())
```

Actions can also collect extra information from the user. For instance, you might have a button to email a client. When the user clicks the button, a modal opens to collect the email subject and body. When the user clicks the "Send" button in the modal, the email is sent:

```
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Facades\Mail;

Action::make('sendEmail')
->form([
    TextInput::make('subject')->required(),
    RichEditor::make('body')->required(),
])
->action(function (array $data) {
    Mail::to($this->client)
        ->send(new GenericEmail(
            subject: $data['subject'],
            body: $data['body'],
        ));
})
```

Usually, actions get executed without redirecting the user away from the page. This is because we extensively use Livewire. However, actions can be much simpler, and don't even need a modal. You can pass a URL to an action, and when the user clicks on the button, they are redirected to that page:

```
Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
```

The entire look of the action's trigger button and the modal is customizable using fluent PHP methods. We provide a sensible and consistent styling for the UI, but all of this is customizable with CSS.

Types of action

The concept of "actions" is used throughout Filament in many contexts. Some contexts don't support opening modals from actions - they can only open a URL, call a public Livewire method, or dispatch a Livewire event. Additionally, different contexts use different action PHP classes since they provide the developer context-aware data that is appropriate to that use-case.

Custom Livewire component actions

You can add an action to any Livewire component in your app, or even a page in a [panel](#).

These actions use the `Filament\Actions\Action` class. They can open a modal if you choose, or even just a URL.

If you're looking to add an action to a Livewire component, [visit this page](#) in the docs. If you want to add an action to the header of a page in a panel, [visit this page](#) instead.

Table actions

Filament's tables also use actions. Actions can be added to the end of any table row, or even in the header of a table. For instance, you may want an action to "create" a new record in the header, and then "edit" and "delete" actions on each row. Additionally, actions can be added to any table column, such that each cell in that column is a trigger for your action.

These actions use the `Filament\Tables\Actions\Action` class. They can open a modal if you choose, or even just a URL.

If you're looking to add an action to a table in your app, [visit this page](#) in the docs.

Table bulk actions

Tables also support "bulk actions". These can be used when the user selects rows in the table. Traditionally, when rows are selected, a "bulk actions" button appears in the top left corner of the table. When the user clicks this button, they are presented with a dropdown menu of actions to choose from. Bulk actions may also be added to the header of a table, next to other header actions. In this case, bulk action trigger buttons are disabled until the user selects table rows.

These actions use the `Filament\Tables\Actions\BulkAction` class. They can open modals if you choose.

If you're looking to add a bulk action to a table in your app, [visit this page](#) in the docs.

Form component actions

Form components can contain actions. A good use case for actions inside form components would be with a select field, and an action button to "create" a new record. When you click on the button, a modal opens to collect the new record's data. When the modal form is submitted, the new record is created in the database, and the select field is filled with the newly created record. Fortunately, [this case is handled for you out of the box](#), but it's a good example of how form component actions can be powerful.

These actions use the `Filament\Forms\Components\Actions\Action` class. They can open a modal if you choose, or even just a URL.

If you're looking to add an action to a form component in your app, [visit this page](#) in the docs.

Infolist component actions

Infolist components can contain actions. These use the `Filament\Infolists\Components\Actions\Action` class. They can open a modal if you choose, or even just a URL.

If you're looking to add an action to an infolist component in your app, [visit this page](#) in the docs.

Notification actions

When you [send notifications](#), you can add actions. These buttons are rendered below the content of the notification. For example, a notification to alert the user that they have a new message should contain an action button that opens the conversation thread.

These actions use the `Filament\Notifications\Actions\Action` class. They aren't able to open modals, but they can open a URL or dispatch a Livewire event.

If you're looking to add an action to a notification in your app, [visit this page](#) in the docs.

Global search result actions

In the Panel Builder, there is a global search field that allows you to search all resources in your app from one place. When you click on a search result, it leads you to the resource page for that record. However, you may add additional actions below each global search result. For example, you may want both "Edit" and "View" options for a client search result, so the user can quickly edit their profile as well as view it in read-only mode.

These actions use the `Filament\GlobalSearch\Actions\Action` class. They aren't able to open modals, but they can open a URL or dispatch a Livewire event.

If you're looking to add an action to a global search result in a panel, [visit this page](#) in the docs.

Prebuilt actions

Filament includes several prebuilt actions that you can add to your app. Their aim is to simplify the most common Eloquent-related actions:

- [Create](#)
- [Edit](#)
- [View](#)
- [Delete](#)
- [Replicate](#)
- [Force-delete](#)
- [Restore](#)
- [Import](#)
- [Export](#)

Grouping actions

You may group actions together into a dropdown menu by using an `ActionGroup` object. Groups may contain many actions, or other groups:

```
ActionGroup::make([
    Action::make('view'),
    Action::make('edit'),
    Action::make('delete'),
])
```

To learn about how to group actions, see the [Grouping actions](#) page.

Trigger Button

Overview

All actions have a trigger button. When the user clicks on it, the action is executed - a modal will open, a closure function will be executed, or they will be redirected to a URL.

This page is about customizing the look of that trigger button.

Choosing a trigger style

Out of the box, action triggers have 4 styles - "button", "link", "icon button", and "badge".

"Button" triggers have a background color, label, and optionally an icon. Usually, this is the default button style, but you can use it manually with the `button()` method:

```
Action::make('edit')  
->button()
```



"Link" triggers have no background color. They must have a label and optionally an icon. They look like a link that you might find embedded within text. You can switch to that style with the `link()` method:

```
Action::make('edit')  
->link()
```



"Icon button" triggers are circular buttons with an icon and no label. You can switch to that style with the `iconButton()` method:

```
Action::make('edit')
->icon('heroicon-m-pencil-square')
->iconButton()
```



"Badge" triggers have a background color, label, and optionally an [icon](#). You can use a badge as trigger using the `badge()` method:

```
Action::make('edit')
->badge()
```

Using an icon button on mobile devices only

You may want to use a button style with a label on desktop, but remove the label on mobile. This will transform it into an icon button. You can do this with the `labeledFrom()` method, passing in the responsive [breakpoint](#) at which you want the label to be added to the button:

```
Action::make('edit')
->icon('heroicon-m-pencil-square')
->button()
->labeledFrom('md')
```

Setting a label

By default, the label of the trigger button is generated from its name. You may customize this using the `label()` method:

```
Action::make('edit')
->label('Edit post')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
```

Optionally, you can have the label automatically translated [using Laravel's localization features](#) with the `translateLabel()` method:

```
Action::make('edit')
->translateLabel() // Equivalent to `label(__('Edit'))`
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
```

Setting a color

Buttons may have a color to indicate their significance. It may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
Action::make('delete')
->color('danger')
```



Delete

Setting a size

Buttons come in 3 sizes - `ActionSize::Small`, `ActionSize::Medium` or `ActionSize::Large`. You can change the size of the action's trigger using the `size()` method:

```
use Filament\Support\Enums\ActionSize;

Action::make('create')
->size(ActionSize::Large)
```



Create

Setting an icon

Buttons may have an `icon` to add more detail to the UI. You can set the icon using the `icon()` method:

```
Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->icon('heroicon-m-pencil-square')
```



You can also change the icon's position to be after the label instead of before it, using the `iconPosition()` method:

```
use Filament\Support\Enums\IconPosition;

Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->icon('heroicon-m-pencil-square')
->iconPosition(IconPosition::After)
```



Authorization

You may conditionally show or hide actions for certain users. To do this, you can use either the `visible()` or `hidden()` methods:

```
Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->visible(auth()->user()->can('update', $this->post))

Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->hidden(! auth()->user()->can('update', $this->post))
```

This is useful for authorization of certain actions to only users who have permission.

Disabling a button

If you want to disable a button instead of hiding it, you can use the `disabled()` method:

```
Action::make('delete')
->disabled()
```

You can conditionally disable a button by passing a boolean to it:

```
Action::make('delete')
->disabled(! auth()->user()->can('delete', $this->post))
```

Registering keybindings

You can attach keyboard shortcuts to trigger buttons. These use the same key codes as [Mousetrap](#):

```
use Filament\Actions\Action;

Action::make('save')
->action(fn () => $this->save())
->keyBindings(['command+s', 'ctrl+s'])
```

Adding a badge to the corner of the button

You can add a badge to the corner of the button, to display whatever you want. It's useful for displaying a count of something, or a status indicator:

```
use Filament\Actions\Action;

Action::make('filter')
->iconButton()
->icon('heroicon-m-funnel')
->badge(5)
```



You can also pass a color to be used for the badge, which can be either `danger`, `gray`, `info`, `primary`, `success` and `warning`:

```
use Filament\Actions\Action;

Action::make('filter')
    ->iconButton()
    ->icon('heroicon-m-funnel')
    ->badge(5)
    ->badgeColor('success')
```



Outlined button style

When you're using the "button" trigger style, you might wish to make it less prominent. You could use a different `color`, but sometimes you might want to make it outlined instead. You can do this with the `outlined()` method:

```
use Filament\Actions\Action;

Action::make('edit')
    ->url(fn(): string => route('posts.edit', ['post' => $this->post]))
    ->button()
    ->outlined()
```

Adding extra HTML attributes

You can pass extra HTML attributes to the button which will be merged onto the outer DOM element. Pass an array of attributes to the `extraAttributes()` method, where the key is the attribute name and the value is the attribute value:

```
use Filament\Actions\Action;

Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->extraAttributes([
    'title' => 'Edit this post',
])
```

If you pass CSS classes in a string, they will be merged with the default classes that already apply to the other HTML element of the button:

```
use Filament\Actions\Action;

Action::make('edit')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
->extraAttributes([
    'class' => 'mx-auto my-8',
])
```

Modals

Overview

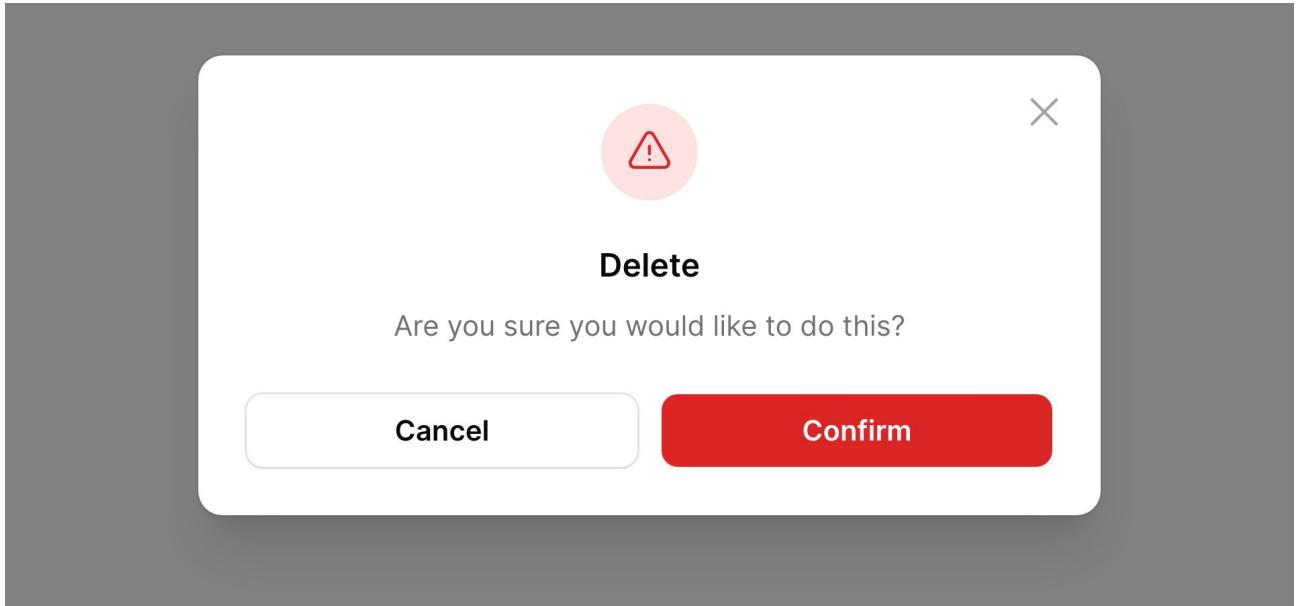
Actions may require additional confirmation or input from the user before they run. You may open a modal before an action is executed to do this.

Confirmation modals

You may require confirmation before an action is run using the `requiresConfirmation()` method. This is useful for particularly destructive actions, such as those that delete records.

```
use App\Models\Post;

Action::make('delete')
    ->action(fn (Post $record) => $record->delete())
    ->requiresConfirmation()
```



The confirmation modal is not available when a `url()` is set instead of an `action()`. Instead, you should redirect to the URL within the `action()` closure.

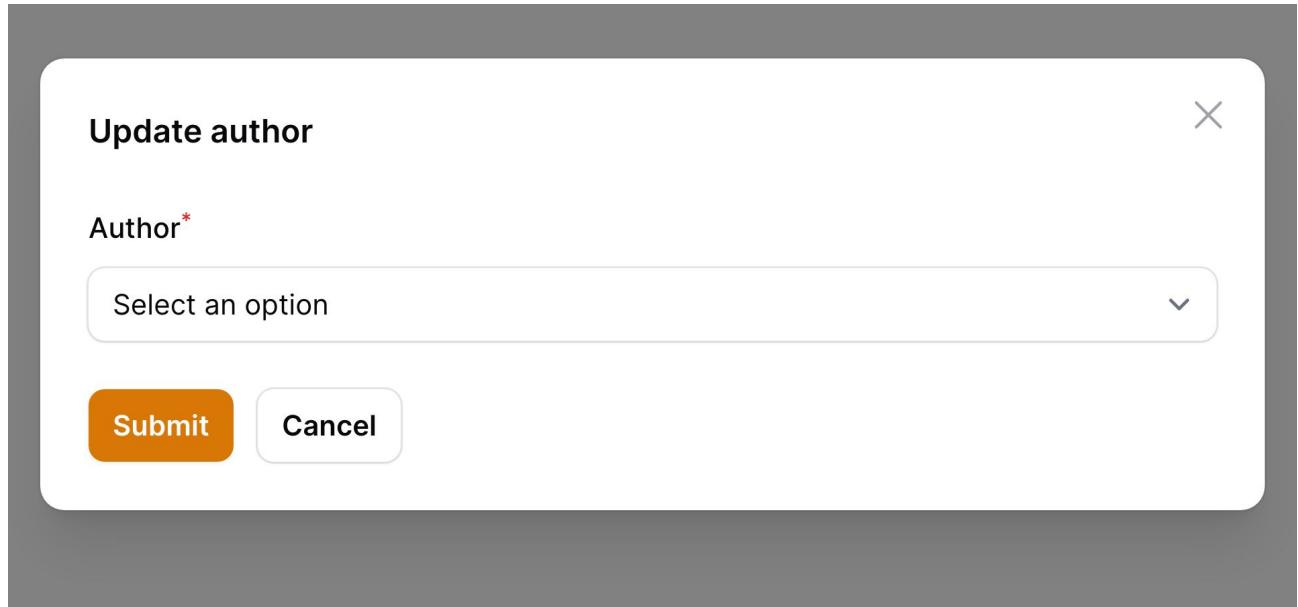
Modal forms

You may also render a form in the modal to collect extra information from the user before the action runs.

You may use components from the [Form Builder](#) to create custom action modal forms. The data from the form is available in the `$data` array of the `action()` closure:

```
use App\Models\Post;
use App\Models\User;
use Filament\Forms\Components>Select;

Action::make('updateAuthor')
->form([
    Select::make('authorId')
        ->label('Author')
        ->options(User::query()->pluck('name', 'id'))
        ->required(),
])
->action(function (array $data, Post $record): void {
    $record->author()->associate($data['authorId']);
    $record->save();
})
```



Filling the form with existing data

You may fill the form with existing data, using the `fillForm()` method:

```
use App\Models\Post;
use App\Models\User;
use Filament\Forms\Components>Select;
use Filament\Forms\Form;

Action::make('updateAuthor')
    ->fillForm(fn (Post $record): array => [
        'authorId' => $record->author->id,
    ])
    ->form([
        Select::make('authorId')
            ->label('Author')
            ->options(User::query()->pluck('name', 'id'))
            ->required(),
    ])
    ->action(function (array $data, Post $record): void {
        $record->author()->associate($data['authorId']);
        $record->save();
    })
)
```

Using a wizard as a modal form

You may create a [multistep form wizard](#) inside a modal. Instead of using a `form()`, define a `steps()` array and pass your `Step` objects:

```

use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Toggle;
use Filament\Forms\Components\Wizard\Step;

Action::make('create')
->steps([
    Step::make('Name')
        ->description('Give the category a unique name')
        ->schema([
            TextInput::make('name')
                ->required()
                ->live()
                ->afterStateUpdated(fn ($state, callable $set) => $set('slug',
Str::slug($state))),
            TextInput::make('slug')
                ->disabled()
                ->required()
                ->unique(Category::class, 'slug'),
        ])
        ->columns(2),
    Step::make('Description')
        ->description('Add some extra details')
        ->schema([
            MarkdownEditor::make('description'),
        ]),
    Step::make('Visibility')
        ->description('Control who can view it')
        ->schema([
            Toggle::make('is_visible')
                ->label('Visible to customers.')
                ->default(true),
        ]),
])
])

```

The screenshot shows a modal window titled "Create" with a "Cancel" button in the top right corner. The window is divided into three horizontal sections, each representing a step in a wizard:

- Step 01: Name**: A text input field labeled "Name" with the placeholder "Give the category unique name".
- Step 02: Description**: A text input field labeled "Description" with the placeholder "Add some extra details".
- Step 03: Visibility**: A toggle switch labeled "Visibility" with the placeholder "Control who can view it".

Below the steps, there are two input fields: "Name*" and "Slug". To the right of the "Name*" field is a "Next" button.

Disabling all form fields

You may wish to disable all form fields in the modal, ensuring the user cannot edit them. You may do so using the `disabledForm()` method:

```
use App\Models\Post;
use App\Models\User;
use Filament\Forms\Components\Textarea;
use Filament\Forms\Components\TextInput;

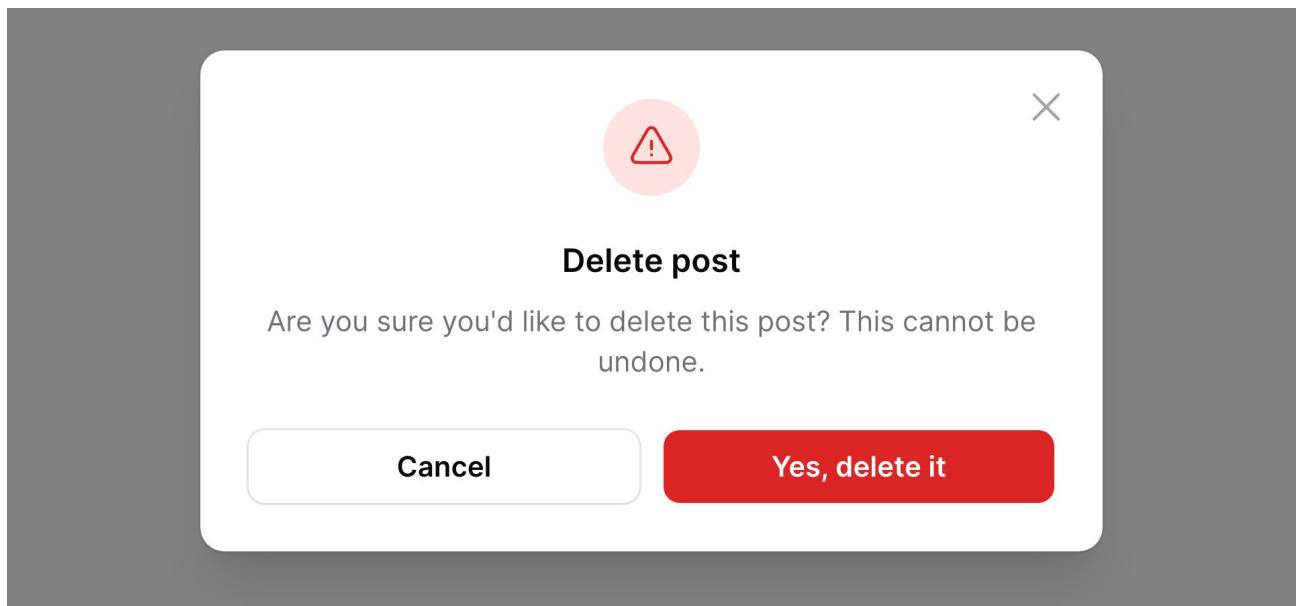
Action::make('approvePost')
    ->form([
        TextInput::make('title'),
        Textarea::make('content'),
    ])
    ->fillForm(fn (Post $record): array => [
        'title' => $record->title,
        'content' => $record->content,
    ])
    ->disabledForm()
    ->action(function (Post $record): void {
        $record->approve();
    })
}
```

Customizing the modal's heading, description, and submit action label

You may customize the heading, description and label of the submit button in the modal:

```
use App\Models\Post;

Action::make('delete')
    ->action(fn (Post $record) => $record->delete())
    ->requiresConfirmation()
    ->modalHeading('Delete post')
    ->modalDescription('Are you sure you\'d like to delete this post? This cannot be undone.')
    ->modalSubmitActionLabel('Yes, delete it')
```

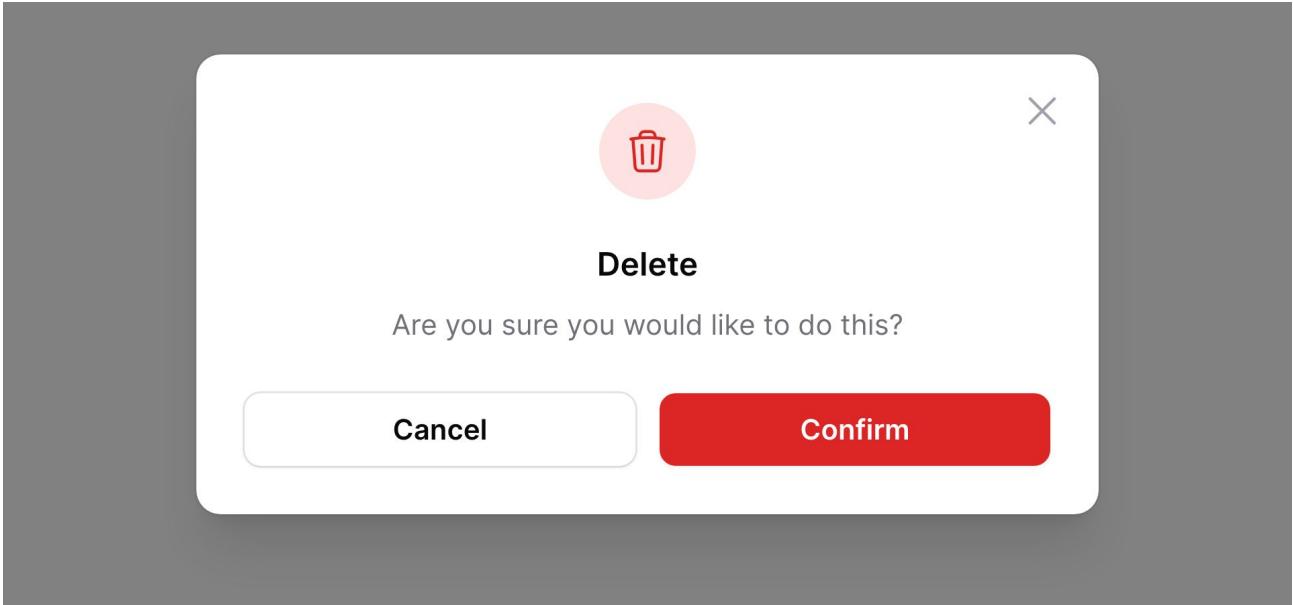


Adding an icon inside the modal

You may add an `icon` inside the modal using the `modalIcon()` method:

```
use App\Models\Post;

Action::make('delete')
    ->action(fn (Post $record) => $record->delete())
    ->requiresConfirmation()
    ->modalIcon('heroicon-o-trash')
```



By default, the icon will inherit the color of the action button. You may customize the color of the icon using the `modalIconColor()` method:

```
use App\Models\Post;

Action::make('delete')
    ->action(fn (Post $record) => $record->delete())
    ->requiresConfirmation()
    ->color('danger')
    ->modalIcon('heroicon-o-trash')
    ->modalIconColor('warning')
```

Customizing the alignment of modal content

By default, modal content will be aligned to the start, or centered if the modal is `xs` or `sm` in `width`. If you wish to change the alignment of content in a modal, you can use the `modalAlignment()` method and pass it `Alignment::Start` or `Alignment::Center`:

```
use Filament\Support\Enums\Alignment;

Action::make('updateAuthor')
    ->form([
        // ...
    ])
    ->action(function (array $data): void {
        // ...
    })
    ->modalAlignment(Alignment::Center)
```

Custom modal content

You may define custom content to be rendered inside your modal, which you can specify by passing a Blade view into the `modalContent()` method:

```
use App\Models\Post;

Action::make('advance')
    ->action(fn (Post $record) => $record->advance())
    ->modalContent(view('filament.pages.actions.advance'))
```

Passing data to the custom modal content

You can pass data to the view by returning it from a function. For example, if the `[$record]` of an action is set, you can pass that through to the view:

```
use Illuminate\Contracts\View\View;

Action::make('advance')
    ->action(fn (Contract $record) => $record->advance())
    ->modalContent(fn (Contract $record): View => view(
        'filament.pages.actions.advance',
        ['record' => $record],
    ))
```

Adding custom modal content below the form

By default, the custom content is displayed above the modal form if there is one, but you can add content below using `modalContentFooter()` if you wish:

```
use App\Models\Post;

Action::make('advance')
    ->action(fn (Post $record) => $record->advance())
    ->modalContentFooter(view('filament.pages.actions.advance'))
```

Adding an action to custom modal content

You can add an action button to your custom modal content, which is useful if you want to add a button that performs an action other than the main action. You can do this by registering an action with the `registerModalActions()` method, and then passing it to the view:

```
use App\Models\Post;
use Illuminate\Contracts\View\View;

Action::make('advance')
    ->registerModalActions([
        Action::make('report')
            ->requiresConfirmation()
            ->action(fn (Post $record) => $record->report()),
    ])
    ->action(fn (Post $record) => $record->advance())
    ->modalContent(fn (Action $action): View => view(
        'filament.pages.actions.advance',
        ['action' => $action],
    )));
})
```

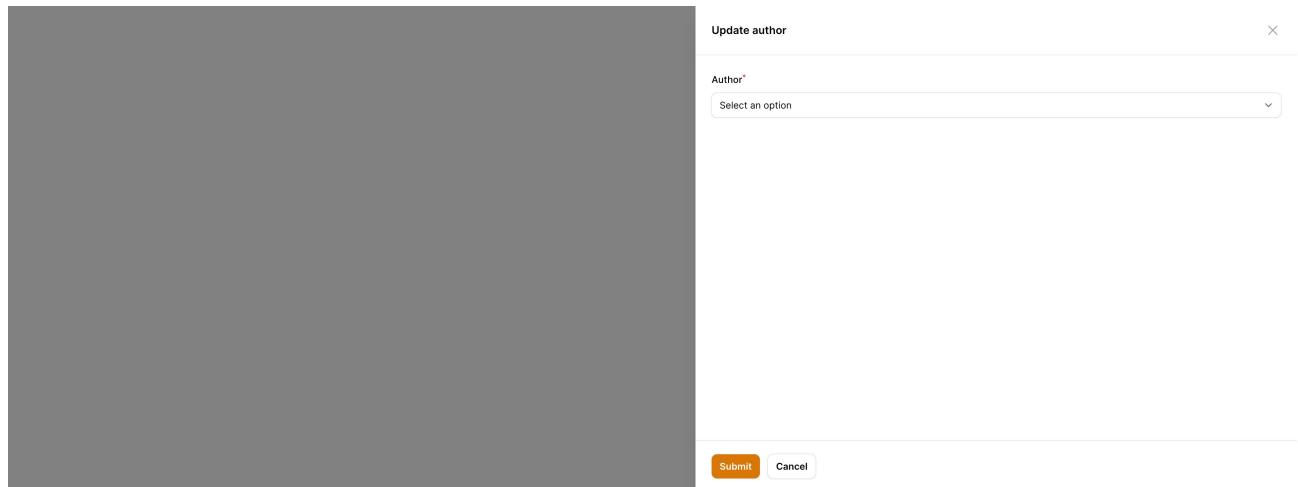
Now, in the view file, you can render the action button by calling `getModalAction()`:

```
<div>
    {{ $action->getModalAction('report') }}
</div>
```

Using a slide-over instead of a modal

You can open a "slide-over" dialog instead of a modal by using the `slideOver()` method:

```
Action::make('updateAuthor')
    ->form([
        // ...
    ])
    ->action(function (array $data): void {
        // ...
    })
    ->slideOver()
```



Instead of opening in the center of the screen, the modal content will now slide in from the right and consume the entire height of the browser.

Making the modal header sticky

The header of a modal scrolls out of view with the modal content when it overflows the modal size. However, slide-overs have a sticky header that's always visible. You may control this behavior using `stickyModalHeader()`:

```
Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->stickyModalHeader()
```

Making the modal footer sticky

The footer of a modal is rendered inline after the content by default. Slide-overs, however, have a sticky footer that always shows when scrolling the content. You may enable this for a modal too using `stickyModalFooter()`:

```
Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->stickyModalFooter()
```

Changing the modal width

You can change the width of the modal by using the `modalWidth()` method. Options correspond to [Tailwind's max-width scale](#). The options are `ExtraSmall`, `Small`, `Medium`, `Large`, `ExtraLarge`, `TwoExtraLarge`, `ThreeExtraLarge`, `FourExtraLarge`, `FiveExtraLarge`, `SixExtraLarge`, `SevenExtraLarge`, and `Screen`:

```
use Filament\Support\Enums\MaxWidth;

Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->modalWidth(MaxWidth::FiveExtraLarge)
```

Executing code when the modal opens

You may execute code within a closure when the modal opens, by passing it to the `mountUsing()` method:

```
use Filament\Forms\Form;

Action::make('create')
->mountUsing(function (Form $form) {
    $form->fill();

    // ...
})
```

The `mountUsing()` method, by default, is used by Filament to initialize the `form`. If you override this method, you will need to call `$form->fill()` to ensure the form is initialized correctly. If you wish to populate the form with data, you can do so by passing an array to the `fill()` method, instead of using `fillForm()` on the action itself.

Customizing the action buttons in the footer of the modal

By default, there are two actions in the footer of a modal. The first is a button to submit, which executes the `action()`. The second button closes the modal and cancels the action.

Modifying a default modal footer action button

To modify the action instance that is used to render one of the default action buttons, you may pass a closure to the `modalSubmitAction()` and `modalCancelAction()` methods:

```
use Filament\Actions\StaticAction;

Action::make('help')
->modalContent(view('actions.help'))
->modalCancelAction(fn (StaticAction $action) => $action->label('Close'))
```

The [methods available to customize trigger buttons](#) will work to modify the `$action` instance inside the closure.

Removing a default modal footer action button

To remove a default action, you may pass `false` to either `modalSubmitAction()` or `modalCancelAction()`:

```
Action::make('help')
->modalContent(view('actions.help'))
->modalSubmitAction(false)
```

Adding an extra modal action button to the footer

You may pass an array of extra actions to be rendered, between the default actions, in the footer of the modal using the `extraModalFooterActions()` method:

```
Action::make('create')
->form([
    // ...
])
// ...
->extraModalFooterActions(fn (Action $action): array => [
    $action->makeModalSubmitAction('createAnother', arguments: ['another' => true]),
])
```

`$action->makeModalSubmitAction()` returns an action instance that can be customized using the [methods available to customize trigger buttons](#).

The second parameter of `makeModalSubmitAction()` allows you to pass an array of arguments that will be accessible inside the action's `action()` closure as `$arguments`. These could be useful as flags to indicate that the action should behave differently based on the user's decision:

```
Action::make('create')
->form([
    // ...
])
// ...
->extraModalFooterActions(fn (Action $action): array => [
    $action->makeModalSubmitAction('createAnother', arguments: ['another' => true]),
])
->action(function (array $data, array $arguments): void {
    // Create

    if ($arguments['another'] ?? false) {
        // Reset the form and don't close the modal
    }
})
})
```

Opening another modal from an extra footer action

You can nest actions within each other, allowing you to open a new modal from an extra footer action:

```
Action::make('edit')
// ...
->extraModalFooterActions([
    Action::make('delete')
        ->requiresConfirmation()
        ->action(function () {
            // ...
        }),
])
```

Now, the edit modal will have a "Delete" button in the footer, which will open a confirmation modal when clicked. This action is completely independent of the `edit` action, and will not run the `edit` action when it is clicked.

In this example though, you probably want to cancel the `edit` action if the `delete` action is run. You can do this using the `cancelParentActions()` method:

```
Action::make('delete')
->requiresConfirmation()
->action(function () {
    // ...
})
->cancelParentActions()
```

If you have deep nesting with multiple parent actions, but you don't want to cancel all of them, you can pass the name of the parent action you want to cancel, including its children, to `cancelParentActions()`:

```

Action::make('first')
->requiresConfirmation()
->action(function () {
    // ...
})
->extraModalFooterActions([
    Action::make('second')
        ->requiresConfirmation()
        ->action(function () {
            // ...
        })
        ->extraModalFooterActions([
            Action::make('third')
                ->requiresConfirmation()
                ->action(function () {
                    // ...
                })
                ->extraModalFooterActions([
                    Action::make('fourth')
                        ->requiresConfirmation()
                        ->action(function () {
                            // ...
                        })
                        ->cancelParentActions('second'),
                ]),
        ]),
])

```

In this example, if the `fourth` action is run, the `second` action is canceled, but so is the `third` action since it is a child of `second`. The `first` action is not canceled, however, since it is the parent of `second`. The `first` action's modal will remain open.

Closing the modal by clicking away

By default, when you click away from a modal, it will close itself. If you wish to disable this behavior for a specific action, you can use the `closeModalByClickingAway(false)` method:

```

Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->closeModalByClickingAway(false)

```

If you'd like to change the behavior for all modals in the application, you can do so by calling `Modal::closedByClickingAway()` inside a service provider or middleware:

```

use Filament\Support\View\Components\Modal;

Modal::closedByClickingAway(false);

```

Closing the modal by escaping

By default, when you press escape on a modal, it will close itself. If you wish to disable this behavior for a specific action, you can use the `closedByEscaping(false)` method:

```
Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->closedByEscaping(false)
```

If you'd like to change the behavior for all modals in the application, you can do so by calling `Modal::closedByEscaping()` inside a service provider or middleware:

```
use Filament\Support\View\Components\Modal;

Modal::closedByEscaping(false);
```

Hiding the modal close button

By default, modals have a close button in the top right corner. If you wish to hide the close button, you can use the `modalCloseButton(false)` method:

```
Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->modalCloseButton(false)
```

If you'd like to hide the close button for all modals in the application, you can do so by calling `Modal::closeButton(false)` inside a service provider or middleware:

```
use Filament\Support\View\Components\Modal;

Modal::closeButton(false);
```

Preventing the modal from autofocusing

By default, modals will autofocus on the first focusable element when opened. If you wish to disable this behavior, you can use the `modalAutofocus(false)` method:

```
Action::make('updateAuthor')
->form([
    // ...
])
->action(function (array $data): void {
    // ...
})
->modalAutofocus(false)
```

If you'd like to disable autofocus for all modals in the application, you can do so by calling `Modal::autofocus(false)` inside a service provider or middleware:

```
use Filament\Support\View\Components\Modal;

Modal::autofocus(false);
```

Optimizing modal configuration methods

When you use database queries or other heavy operations inside modal configuration methods like `modalHeading()`, they can be executed more than once. This is because Filament uses these methods to decide whether to render the modal or not, and also to render the modal's content.

To skip the check that Filament does to decide whether to render the modal, you can use the `modal()` method, which will inform Filament that the modal exists for this action and it does not need to check again:

```
Action::make('updateAuthor')
->modal()
```

Conditionally hiding the modal

You may have a need to conditionally show a modal for confirmation reasons while falling back to the default action. This can be achieved using `modalHidden()`:

```
Action::make('create')
->action(function (array $data): void {
    // ...
})
->modalHidden(fn (): bool => $this->role !== 'admin')
->modalContent(view('filament.pages.actions.create'))
```

Adding extra attributes to the modal window

You may also pass extra HTML attributes to the modal window using `extraModalWindowAttributes()`:

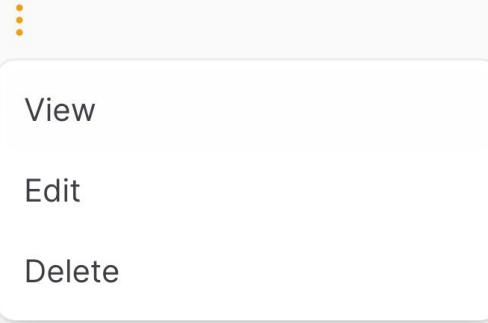
```
Action::make('updateAuthor')
->extraModalWindowAttributes(['class' => 'update-author-modal'])
```

Grouping Actions

Overview

You may group actions together into a dropdown menu by using an `ActionGroup` object. Groups may contain many actions, or other groups:

```
ActionGroup::make([
    Action::make('view'),
    Action::make('edit'),
    Action::make('delete'),
])
```



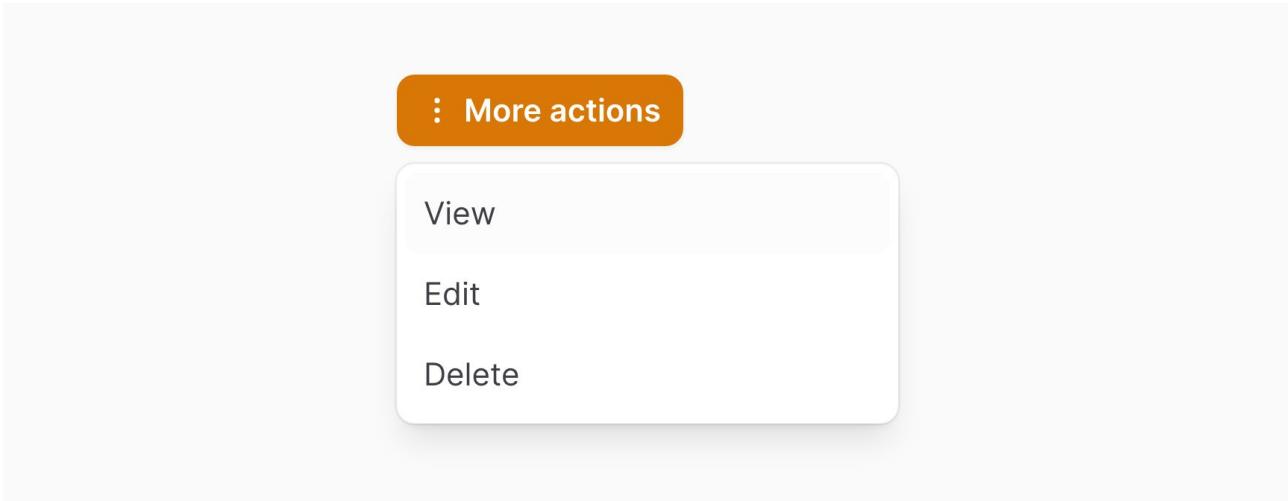
This page is about customizing the look of the group's trigger button and dropdown.

Customizing the group trigger style

The button which opens the dropdown may be customized in the same way as a normal action. [All the methods available for trigger buttons](#) may be used to customize the group trigger button:

```
use Filament\Support\Enums\ActionSize;

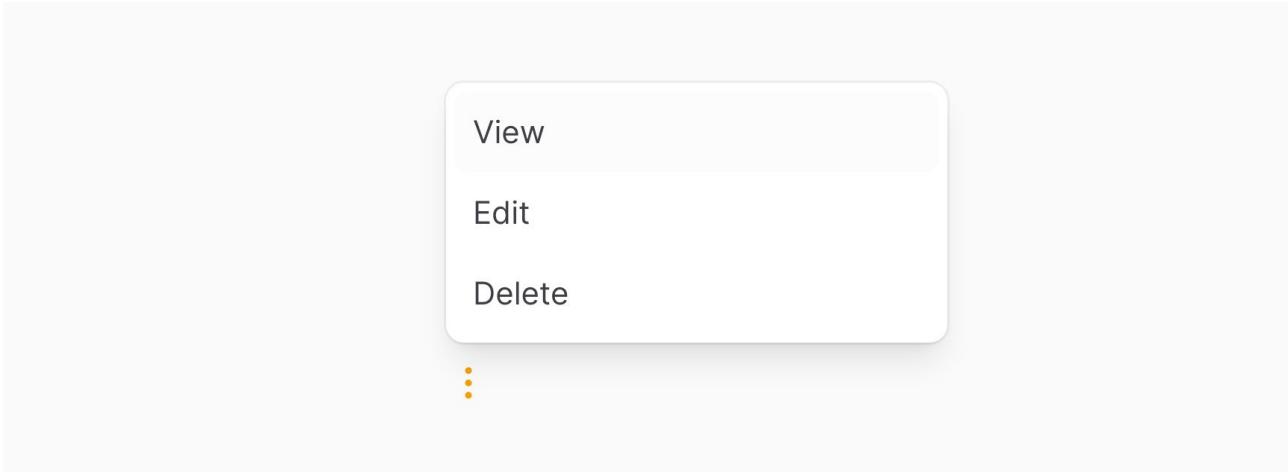
ActionGroup::make([
    // Array of actions
])
->label('More actions')
->icon('heroicon-m-ellipsis-vertical')
->size(ActionSize::Small)
->color('primary')
->button()
```



Setting the placement of the dropdown

The dropdown may be positioned relative to the trigger button by using the `dropdownPlacement()` method:

```
ActionGroup::make([
    // Array of actions
])
->dropdownPlacement('top-start')
```

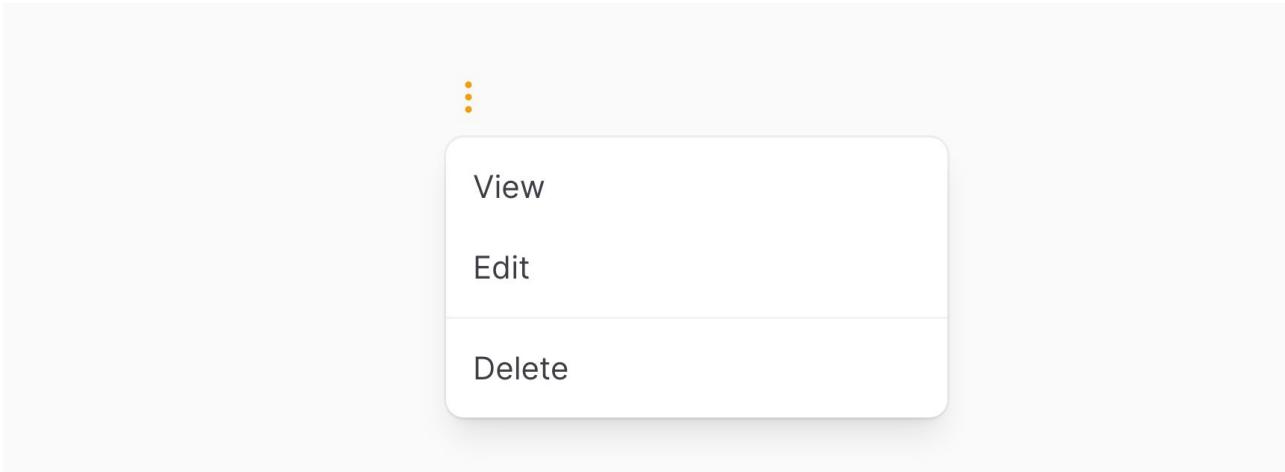


Adding dividers between actions

You may add dividers between groups of actions by using nested `ActionGroup` objects:

```
ActionGroup::make([
    ActionGroup::make([
        // Array of actions
    ])
->dropdown(false),
    // Array of actions
])
```

The `dropdown(false)` method puts the actions inside the parent dropdown, instead of a new nested dropdown.



Setting the width of the dropdown

The dropdown may be set to a width by using the `dropdownWidth()` method. Options correspond to [Tailwind's max-width scale](#). The options are `ExtraSmall`, `Small`, `Medium`, `Large`, `ExtraLarge`, `TwoExtraLarge`, `ThreeExtraLarge`, `FourExtraLarge`, `FiveExtraLarge`, `SixExtraLarge` and `SevenExtraLarge`:

```
use Filament\Support\Enums\MaxWidth;

ActionGroup::make([
    // Array of actions
])
->dropdownWidth(MaxWidth::ExtraSmall)
```

Controlling the maximum height of the dropdown

The dropdown content can have a maximum height using the `maxHeight()` method, so that it scrolls. You can pass a [CSS length](#):

```
ActionGroup::make([
    // Array of actions
])
->maxHeight('400px')
```

Controlling the dropdown offset

You may control the offset of the dropdown using the `dropdownOffset()` method, by default the offset is set to `8`.

```
ActionGroup::make([
    // Array of actions
])
->dropdownOffset(16)
```

Adding An Action To A Livewire Component

Setting up the Livewire component

First, generate a new Livewire component:

```
php artisan make:livewire ManageProduct
```

Then, render your Livewire component on the page:

```
@livewire('manage-product')
```

Alternatively, you can use a full-page Livewire component:

```
use App\LiveWire\ManageProduct;
use Illuminate\Support\Facades\Route;

Route::get('products/{product}/manage', ManageProduct::class);
```

You must use the `InteractsWithActions` and `InteractsWithForms` traits, and implement the `HasActions` and `HasForms` interfaces on your Livewire component class:

```
use Filament\Actions\Concerns\InteractsWithActions;
use Filament\Actions\Contracts\HasActions;
use Filament\Forms\Concerns\InteractsWithForms;
use Filament\Forms\Contracts\HasForms;
use LiveWire\Component;

class ManagePost extends Component implements HasForms, HasActions
{
    use InteractsWithActions;
    use InteractsWithForms;

    // ...
}
```

Adding the action

Add a method that returns your action. The method must share the exact same name as the action, or the name followed by `Action`:

```

use App\Models\Post;
use Filament\Actions\Action;
use Filament\Actions\Concerns\InteractsWithActions;
use Filament\Actions\Contracts\HasActions;
use Filament\Forms\Concerns\InteractsWithForms;
use Filament\Forms\Contracts\HasForms;
use Livewire\Component;

class ManagePost extends Component implements HasForms, HasActions
{
    use InteractsWithActions;
    use InteractsWithForms;

    public Post $post;

    public function deleteAction(): Action
    {
        return Action::make('delete')
            ->requiresConfirmation()
            ->action(fn () => $this->post->delete());
    }

    // This method name also works, since the action name is `delete`:
    // public function delete(): Action

    // This method name does not work, since the action name is `delete`, not `deletePost`:
    // public function deletePost(): Action

    // ...
}

```

Finally, you need to render the action in your view. To do this, you can use `{{ $this->deleteAction }}`, where you replace `deleteAction` with the name of your action method:

```

<div>
    {{ $this->deleteAction }}

    <x-filament-actions::modals />
</div>

```

You also need `<x-filament-actions::modals />` which injects the HTML required to render action modals. This only needs to be included within the Livewire component once, regardless of how many actions you have for that component.

Passing action arguments

Sometimes, you may wish to pass arguments to your action. For example, if you're rendering the same action multiple times in the same view, but each time for a different model, you could pass the model ID as an argument, and then retrieve it later. To do this, you can invoke the action in your view and pass in the arguments as an array:

```
<div>
    @foreach ($posts as $post)
        <h2>{{ $post->title }}</h2>

        {{ ($this->deleteAction)(['post' => $post->id]) }}
    @endforeach

    <x-filament-actions::modals />
</div>
```

Now, you can access the post ID in your action method:

```
use App\Models\Post;
use Filament\Actions\Action;

public function deleteAction(): Action
{
    return Action::make('delete')
        ->requiresConfirmation()
        ->action(function (array $arguments) {
            $post = Post::find($arguments['post']);

            $post?->delete();
        });
}
```

Hiding actions in a Livewire view

If you use `hidden()` or `visible()` to control if an action is rendered, you should wrap the action in an `@if` check for `isVisible()`:

```
<div>
    @if ($this->deleteAction->isVisible())
        {{ $this->deleteAction }}
    @endif

    {{-- Or --}}


    @if ((($this->deleteAction)(['post' => $post->id])->isVisible()))
        {{ ($this->deleteAction)(['post' => $post->id]) }}
    @endif
</div>
```

The `hidden()` and `visible()` methods also control if the action is `disabled()`, so they are still useful to protect the action from being run if the user does not have permission. Encapsulating this logic in the `hidden()` or `visible()` of the action itself is good practice otherwise you need to define the condition in the view and in `disabled()`.

You can also take advantage of this to hide any wrapping elements that may not need to be rendered if the action is hidden:

```
<div>
  @if ($this->deleteAction->isVisible())
    <div>
      {{ $this->deleteAction }}
    </div>
  @endif
</div>
```

Grouping actions in a Livewire view

You may group actions together into a dropdown menu by using the `<x-filament-actions::group>` Blade component, passing in the `actions` array as an attribute:

```
<div>
  <x-filament-actions::group :actions="[
    $this->editAction,
    $this->viewAction,
    $this->deleteAction,
  ]" />

  <x-filament-actions::modals />
</div>
```

You can also pass in any attributes to customize the appearance of the trigger button and dropdown:

```
<div>
  <x-filament-actions::group
    :actions="[
      $this->editAction,
      $this->viewAction,
      $this->deleteAction,
    ]"
    label="Actions"
    icon="heroicon-m-ellipsis-vertical"
    color="primary"
    size="md"
    tooltip="More actions"
    dropdown-placement="bottom-start"
  />

  <x-filament-actions::modals />
</div>
```

Chaining actions

You can chain multiple actions together, by calling the `replaceMountedAction()` method to replace the current action with another when it has finished:

```

use App\Models\Post;
use Filament\Actions\Action;

public function editAction(): Action
{
    return Action::make('edit')
        ->form([
            // ...
        ])
        // ...
        ->action(function (array $arguments) {
            $post = Post::find($arguments['post']);

            // ...

            $this->replaceMountedAction('publish', $arguments);
        });
}

public function publishAction(): Action
{
    return Action::make('publish')
        ->requiresConfirmation()
        // ...
        ->action(function (array $arguments) {
            $post = Post::find($arguments['post']);

            $post->publish();
        });
}

```

Now, when the first action is submitted, the second action will open in its place. The `arguments` that were originally passed to the first action get passed to the second action, so you can use them to persist data between requests.

If the first action is canceled, the second one is not opened. If the second action is canceled, the first one has already run and cannot be cancelled.

Programmatically triggering actions

Sometimes you may need to trigger an action without the user clicking on the built-in trigger button, especially from JavaScript. Here is an example action which could be registered on a Livewire component:

```

use Filament\Actions\Action;

public function testAction(): Action
{
    return Action::make('test')
        ->requiresConfirmation()
        ->action(function (array $arguments) {
            dd('Test action called', $arguments);
        });
}

```

You can trigger that action from a click in your HTML using the `wire:click` attribute, calling the `mountAction()` method and optionally passing in any arguments that you want to be available:

```
<button wire:click="mountAction('test', { id: 12345 })">  
  Button  
</button>
```

To trigger that action from JavaScript, you can use the `$wire` utility, passing in the same arguments:

```
$wire.mountAction('test', { id: 12345 })
```

Prebuilt Actions

Create

Overview

Filament includes a prebuilt action that is able to create Eloquent records. When the trigger button is clicked, a modal will open with a form inside. The user fills the form, and that data is validated and saved into the database. You may use it like so:

```
use Filament\Actions\CreateAction;
use Filament\Forms\Components\TextInput;

CreateAction::make()
    ->model(Post::class)
    ->form([
        TextInput::make('title')
            ->required()
            ->maxLength(255),
        // ...
    ])
)
```

If you want to add this action to the header of a table instead, you can use `Filament\Tables\Actions\CreateAction`:

```
use Filament\Forms\Components\TextInput;
use Filament\Tables\Actions\CreateAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->headerActions([
            CreateAction::make()
                ->form([
                    TextInput::make('title')
                        ->required()
                        ->maxLength(255),
                    // ...
                ]),
        ]);
}
```

Customizing data before saving

Sometimes, you may wish to modify form data before it is finally saved to the database. To do this, you may use the `mutateFormDataUsing()` method, which has access to the `$data` as an array, and returns the modified version:

```
CreateAction::make()
->mutateFormDataUsing(function (array $data): array {
    $data['user_id'] = auth()->id();

    return $data;
})
```

Customizing the creation process

You can tweak how the record is created with the `using()` method:

```
use Illuminate\Database\Eloquent\Model;

CreateAction::make()
->using(function (array $data, string $model): Model {
    return $model::create($data);
})
```

`$model` is the class name of the model, but you can replace this with your own hard-coded class if you wish.

Redirecting after creation

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
CreateAction::make()
->successRedirectUrl(route('posts.list'))
```

If you want to redirect using the created record, use the `$record` parameter:

```
use Illuminate\Database\Eloquent\Model;

CreateAction::make()
->successRedirectUrl(fn (Model $record): string => route('posts.edit', [
    'post' => $record,
]))
```

Customizing the save notification

When the record is successfully created, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
CreateAction::make()
->successNotificationTitle('User registered')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;

CreateAction::make()
    ->successNotification(
        Notification::make()
            ->success()
            ->title('User registered')
            ->body('The user has been created successfully.'),
    )
)
```

To disable the notification altogether, use the `successNotification(null)` method:

```
CreateAction::make()
    ->successNotification(null)
```

Lifecycle hooks

Hooks may be used to execute code at various points within the action's lifecycle, like before a form is saved.

There are several available hooks:

```
CreateAction::make()
    ->beforeFormFilled(function () {
        // Runs before the form fields are populated with their default values.
    })
    ->afterFormFilled(function () {
        // Runs after the form fields are populated with their default values.
    })
    ->beforeFormValidated(function () {
        // Runs before the form fields are validated when the form is submitted.
    })
    ->afterFormValidated(function () {
        // Runs after the form fields are validated when the form is submitted.
    })
    ->before(function () {
        // Runs before the form fields are saved to the database.
    })
    ->after(function () {
        // Runs after the form fields are saved to the database.
    })
})
```

Halting the creation process

At any time, you may call `$action->halt()` from inside a lifecycle hook or mutation method, which will halt the entire creation process:

```

use App\Models\Post;
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

CreateAction::make()
    ->before(function (CreateAction $action, Post $record) {
        if (! $record->team->subscribed()) {
            Notification::make()
                ->warning()
                ->title('You don\'t have an active subscription!')
                ->body('Choose a plan to continue.')
                ->persistent()
                ->actions([
                    Action::make('subscribe')
                        ->button()
                        ->url(route('subscribe'), shouldOpenInNewTab: true),
                ])
                ->send();
        }

        $action->halt();
    })
}
)

```

If you'd like the action modal to close too, you can completely `cancel()` the action instead of halting it:

```
$action->cancel();
```

Using a wizard

You may easily transform the creation process into a multistep wizard. Instead of using a `form()`, define a `steps()` array and pass your `Step` objects:

```

use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Toggle;
use Filament\Forms\Components\Wizard\Step;

CreateAction::make()
->steps([
    Step::make('Name')
        ->description('Give the category a unique name')
        ->schema([
            TextInput::make('name')
                ->required()
                ->live()
                ->afterStateUpdated(fn ($state, callable $set) => $set('slug',
Str::slug($state))),
            TextInput::make('slug')
                ->disabled()
                ->required()
                ->unique(Category::class, 'slug'),
        ])
        ->columns(2),
    Step::make('Description')
        ->description('Add some extra details')
        ->schema([
            MarkdownEditor::make('description'),
        ]),
    Step::make('Visibility')
        ->description('Control who can view it')
        ->schema([
            Toggle::make('is_visible')
                ->label('Visible to customers.')
                ->default(true),
        ]),
])
])

```

Now, create a new record to see your wizard in action! Edit will still use the form defined within the resource class.

If you'd like to allow free navigation, so all the steps are skippable, use the `skippableSteps()` method:

```

CreateAction::make()
->steps([
    // ...
])
->skippableSteps()

```

Disabling create another

If you'd like to remove the "create another" button from the modal, you can use the `createAnother(false)` method:

```

CreateAction::make()
->createAnother(false)

```

Edit

Overview

Filament includes a prebuilt action that is able to edit Eloquent records. When the trigger button is clicked, a modal will open with a form inside. The user fills the form, and that data is validated and saved into the database. You may use it like so:

```
use Filament\Actions>EditAction;
use Filament\Forms\Components\TextInput;

EditAction::make()
    ->record($this->post)
    ->form([
        TextInput::make('title')
            ->required()
            ->maxLength(255),
        // ...
    ])
)
```

If you want to edit table rows, you can use the `Filament\Tables\Actions>EditAction` instead:

```
use Filament\Forms\Components\TextInput;
use Filament\Tables\Actions>EditAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            EditAction::make()
                ->form([
                    TextInput::make('title')
                        ->required()
                        ->maxLength(255),
                    // ...
                ]),
        ]);
}
```

Customizing data before filling the form

You may wish to modify the data from a record before it is filled into the form. To do this, you may use the `mutateRecordDataUsing()` method to modify the `$data` array, and return the modified version before it is filled into the form:

```
EditAction::make()
->mutateRecordDataUsing(function (array $data): array {
    $data['user_id'] = auth()->id();

    return $data;
})
```

Customizing data before saving

Sometimes, you may wish to modify form data before it is finally saved to the database. To do this, you may use the `mutateFormDataUsing()` method, which has access to the `$data` as an array, and returns the modified version:

```
EditAction::make()
->mutateFormDataUsing(function (array $data): array {
    $data['last_edited_by_id'] = auth()->id();

    return $data;
})
```

Customizing the saving process

You can tweak how the record is updated with the `using()` method:

```
use Illuminate\Database\Eloquent\Model;

EditAction::make()
->using(function (Model $record, array $data): Model {
    $record->update($data);

    return $record;
})
```

Redirecting after saving

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
EditAction::make()
->successRedirectUrl(route('posts.list'))
```

If you want to redirect using the created record, use the `$record` parameter:

```
use Illuminate\Database\Eloquent\Model;

EditAction::make()
->successRedirectUrl(fn (Model $record): string => route('posts.view', [
    'post' => $record,
]))
```

Customizing the save notification

When the record is successfully updated, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
EditAction::make()
->successNotificationTitle('User updated')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;

EditAction::make()
->successNotification(
    Notification::make()
        ->success()
        ->title('User updated')
        ->body('The user has been saved successfully.'))
)
```

To disable the notification altogether, use the `successNotification(null)` method:

```
EditAction::make()
->successNotification(null)
```

Lifecycle hooks

Hooks may be used to execute code at various points within the action's lifecycle, like before a form is saved.

There are several available hooks:

```
EditAction::make()
->beforeFormFilled(function () {
    // Runs before the form fields are populated from the database.
})
->afterFormFilled(function () {
    // Runs after the form fields are populated from the database.
})
->beforeFormValidated(function () {
    // Runs before the form fields are validated when the form is saved.
})
->afterFormValidated(function () {
    // Runs after the form fields are validated when the form is saved.
})
->before(function () {
    // Runs before the form fields are saved to the database.
})
->after(function () {
    // Runs after the form fields are saved to the database.
})
```

Halting the saving process

At any time, you may call `$action->halt()` from inside a lifecycle hook or mutation method, which will halt the entire saving process:

```
use App\Models\Post;
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;
use Filament\Tables\Actions>EditAction;

EditAction::make()
    ->before(function (EditAction $action, Post $record) {
        if (! $record->team->subscribed()) {
            Notification::make()
                ->warning()
                ->title('You don\'t have an active subscription!')
                ->body('Choose a plan to continue.')
                ->persistent()
                ->actions([
                    Action::make('subscribe')
                        ->button()
                        ->url(route('subscribe'), shouldOpenInNewTab: true),
                ])
            ->send();
        }

        $action->halt();
    }
})
```

If you'd like the action modal to close too, you can completely `cancel()` the action instead of halting it:

```
$action->cancel();
```

View

Overview

Filament includes a prebuilt action that is able to view Eloquent records. When the trigger button is clicked, a modal will open with information inside. Filament uses form fields to structure this information. All form fields are disabled, so they are not editable by the user. You may use it like so:

```
use Filament\Actions\ViewAction;
use Filament\Forms\Components\TextInput;

ViewAction::make()
    ->record($this->post)
    ->form([
        TextInput::make('title')
            ->required()
            ->maxLength(255),
        // ...
    ])
)
```

If you want to view table rows, you can use the `Filament\Tables\Actions\ViewAction` instead:

```
use Filament\Forms\Components\TextInput;
use Filament\Tables\Actions\ViewAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            ViewAction::make()
                ->form([
                    TextInput::make('title')
                        ->required()
                        ->maxLength(255),
                    // ...
                ]),
        ]);
}
```

Customizing data before filling the form

You may wish to modify the data from a record before it is filled into the form. To do this, you may use the `mutateRecordDataUsing()` method to modify the `$data` array, and return the modified version before it is filled into the form:

```
ViewAction::make()  
->mutateRecordDataUsing(function (array $data): array {  
    $data['user_id'] = auth()->id();  
  
    return $data;  
})
```

Delete

Overview

Filament includes a prebuilt action that is able to delete Eloquent records. When the trigger button is clicked, a modal asks the user for confirmation. You may use it like so:

```
use Filament\Actions\DeleteAction;

DeleteAction::make()
    ->record($this->post)
```

If you want to delete table rows, you can use the `Filament\Tables\Actions\DeleteAction` instead, or `Filament\Tables\Actions\DeleteBulkAction` to delete multiple at once:

```
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Actions\DeleteAction;
use Filament\Tables\Actions\DeleteBulkAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            DeleteAction::make(),
            // ...
        ])
        ->bulkActions([
            BulkActionGroup::make([
                DeleteBulkAction::make(),
                // ...
            ]),
        ]);
}
```

Redirecting after deleting

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
DeleteAction::make()
    ->successRedirectUrl(route('posts.list'))
```

Customizing the delete notification

When the record is successfully deleted, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
DeleteAction::make()  
    ->successNotificationTitle('User deleted')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;  
  
DeleteAction::make()  
    ->successNotification(  
        Notification::make()  
            ->success()  
            ->title('User deleted')  
            ->body('The user has been deleted successfully.'),  
    )
```

To disable the notification altogether, use the `successNotification(null)` method:

```
DeleteAction::make()  
    ->successNotification(null)
```

Lifecycle hooks

You can use the `before()` and `after()` methods to execute code before and after a record is deleted:

```
DeleteAction::make()  
    ->before(function () {  
        // ...  
    })  
    ->after(function () {  
        // ...  
    })
```

Replicate

Overview

Filament includes a prebuilt action that is able to `replicate` Eloquent records. You may use it like so:

```
use Filament\Actions\ReplicateAction;

ReplicateAction::make()
->record($this->post)
```

If you want to replicate table rows, you can use the `Filament\Tables\Actions\ReplicateAction` instead:

```
use Filament\Tables\Actions\ReplicateAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            ReplicateAction::make(),
            // ...
        ]);
}
```

Excluding attributes

The `excludeAttributes()` method is used to instruct the action which columns should be excluded from replication:

```
ReplicateAction::make()
->excludeAttributes(['slug'])
```

Customizing data before filling the form

You may wish to modify the data from a record before it is filled into the form. To do this, you may use the `mutateRecordDataUsing()` method to modify the `$data` array, and return the modified version before it is filled into the form:

```
ReplicateAction::make()
->mutateRecordDataUsing(function (array $data): array {
    $data['user_id'] = auth()->id();

    return $data;
})
```

Redirecting after replication

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
ReplicateAction::make()
->successRedirectUrl(route('posts.list'))
```

If you want to redirect using the replica, use the `[$replica]` parameter:

```
use Illuminate\Database\Eloquent\Model;

ReplicateAction::make()
->successRedirectUrl(fn (Model $replica): string => route('posts.edit', [
    'post' => $replica,
]))
```

Customizing the replicate notification

When the record is successfully replicated, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
ReplicateAction::make()
->successNotificationTitle('Category replicated')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;

ReplicateAction::make()
->successNotification(
    Notification::make()
        ->success()
        ->title('Category replicated')
        ->body('The category has been replicated successfully.'),
)
```

Lifecycle hooks

Hooks may be used to execute code at various points within the action's lifecycle, like before the replica is saved.

```
use Illuminate\Database\Eloquent\Model;

ReplicateAction::make()
->before(function () {
    // Runs before the record has been replicated.
})
->beforeReplicaSaved(function (Model $replica): void {
    // Runs after the record has been replicated but before it is saved to the database.
})
->after(function (Model $replica): void {
    // Runs after the replica has been saved to the database.
})
```

Halting the replication process

At any time, you may call `$action->halt()` from inside a lifecycle hook, which will halt the entire replication process:

```
use App\Models\Post;
use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;

ReplicateAction::make()
    ->before(function (ReplicateAction $action, Post $record) {
        if (! $record->team->subscribed()) {
            Notification::make()
                ->warning()
                ->title('You don\'t have an active subscription!')
                ->body('Choose a plan to continue.')
                ->persistent()
                ->actions([
                    Action::make('subscribe')
                        ->button()
                        ->url(route('subscribe'), shouldOpenInNewTab: true),
                ])
                ->send();
        }

        $action->halt();
    })
})
```

If you'd like the action modal to close too, you can completely `cancel()` the action instead of halting it:

```
$action->cancel();
```

Force Delete

Overview

Filament includes a prebuilt action that is able to force-delete soft deleted Eloquent records. When the trigger button is clicked, a modal asks the user for confirmation. You may use it like so:

```
use Filament\Actions\ForceDeleteAction;

ForceDeleteAction::make()
    ->record($this->post)
```

If you want to force-delete table rows, you can use the `Filament\Tables\Actions\ForceDeleteAction` instead, or `Filament\Tables\Actions\ForceDeleteBulkAction` to force-delete multiple at once:

```
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Actions\ForceDeleteAction;
use Filament\Tables\Actions\ForceDeleteBulkAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            ForceDeleteAction::make(),
            // ...
        ])
        ->bulkActions([
            BulkActionGroup::make([
                ForceDeleteBulkAction::make(),
                // ...
            ]),
        ]);
}
```

Redirecting after force-deleting

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
ForceDeleteAction::make()
    ->successRedirectUrl(route('posts.list'))
```

Customizing the force-delete notification

When the record is successfully force-deleted, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
ForceDeleteAction::make()  
    ->successNotificationTitle('User force-deleted')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;  
  
ForceDeleteAction::make()  
    ->successNotification(  
        Notification::make()  
            ->success()  
            ->title('User force-deleted')  
            ->body('The user has been force-deleted successfully.'),  
    )
```

To disable the notification altogether, use the `successNotification(null)` method:

```
ForceDeleteAction::make()  
    ->successNotification(null)
```

Lifecycle hooks

You can use the `before()` and `after()` methods to execute code before and after a record is force-deleted:

```
ForceDeleteAction::make()  
    ->before(function () {  
        // ...  
    })  
    ->after(function () {  
        // ...  
    })
```

Restore

Overview

Filament includes a prebuilt action that is able to restore soft deleted Eloquent records. When the trigger button is clicked, a modal asks the user for confirmation. You may use it like so:

```
use Filament\Actions\RestoreAction;

RestoreAction::make()
    ->record($this->post)
```

If you want to restore table rows, you can use the `Filament\Tables\Actions\RestoreAction` instead, or `Filament\Tables\Actions\RestoreBulkAction` to restore multiple at once:

```
use Filament\Tables\Actions\BulkActionGroup;
use Filament\Tables\Actions\RestoreAction;
use Filament\Tables\Actions\RestoreBulkAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->actions([
            RestoreAction::make(),
            // ...
        ])
        ->bulkActions([
            BulkActionGroup::make([
                RestoreBulkAction::make(),
                // ...
            ]),
        ]);
}
```

Redirecting after restoring

You may set up a custom redirect when the form is submitted using the `successRedirectUrl()` method:

```
RestoreAction::make()
    ->successRedirectUrl(route('posts.list'))
```

Customizing the restore notification

When the record is successfully restored, a notification is dispatched to the user, which indicates the success of their action.

To customize the title of this notification, use the `successNotificationTitle()` method:

```
RestoreAction::make()  
    ->successNotificationTitle('User restored')
```

You may customize the entire notification using the `successNotification()` method:

```
use Filament\Notifications\Notification;  
  
RestoreAction::make()  
    ->successNotification(  
        Notification::make()  
            ->success()  
            ->title('User restored')  
            ->body('The user has been restored successfully.'),  
    )
```

To disable the notification altogether, use the `successNotification(null)` method:

```
RestoreAction::make()  
    ->successNotification(null)
```

Lifecycle hooks

You can use the `before()` and `after()` methods to execute code before and after a record is restored:

```
RestoreAction::make()  
    ->before(function () {  
        // ...  
    })  
    ->after(function () {  
        // ...  
    })
```

Import

Overview

Filament v3.1 introduced a prebuilt action that is able to import rows from a CSV. When the trigger button is clicked, a modal asks the user for a file. Once they upload one, they are able to map each column in the CSV to a real column in the database. If any rows fail validation, they will be compiled into a downloadable CSV for the user to review after the rest of the rows have been imported. Users can also download an example CSV file containing all the columns that can be imported.

This feature uses [job batches](#) and [database notifications](#), so you need to publish those migrations from Laravel. Also, you need to publish the migrations for tables that Filament uses to store information about imports:

```
# Laravel 11 and higher
php artisan make:queue-batches-table
php artisan make:notifications-table

# Laravel 10
php artisan queue:batches-table
php artisan notifications:table
```

```
# All apps
php artisan vendor:publish --tag=filament-actions-migrations
php artisan migrate
```

If you're using PostgreSQL, make sure that the `[data]` column in the notifications migration is using `[json()]`:
`$table->json('data');`

If you're using UUIDs for your `[User]` model, make sure that your `[notifiable]` column in the notifications migration is using `[uuidMorphs()]`:
`$table->uuidMorphs('notifiable');`

You may use the `[ImportAction]` like so:

```
use App\Filament\Imports\ProductImporter;
use Filament\Actions\ImportAction;

ImportAction::make()
    ->importer(ProductImporter::class)
```

If you want to add this action to the header of a table instead, you can use `[Filament\Tables\Actions\ImportAction]`:

```

use App\Filament\Imports\ProductImporter;
use Filament\Tables\Actions\ImportAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->headerActions([
            ImportAction::make()
                ->importer(ProductImporter::class)
        ]);
}

```

The "importer" class needs to be created to tell Filament how to import each row of the CSV.

If you have more than one `ImportAction` in the same place, you should give each a unique name in the `make()` method:

```

ImportAction::make('importProducts')
    ->importer(ProductImporter::class)

ImportAction::make('importBrands')
    ->importer(BrandImporter::class)

```

Creating an importer

To create an importer class for a model, you may use the `make:filament-importer` command, passing the name of a model:

```
php artisan make:filament-importer Product
```

This will create a new class in the `app/Filament/Imports` directory. You now need to define the columns that can be imported.

Automatically generating importer columns

If you'd like to save time, Filament can automatically generate the columns for you, based on your model's database columns, using `--generate`:

```
php artisan make:filament-importer Product --generate
```

Defining importer columns

To define the columns that can be imported, you need to override the `getColumns()` method on your importer class, returning an array of `ImportColumn` objects:

```
use Filament\Actions\Imports\ImportColumn;

public static function getColumns(): array
{
    return [
        ImportColumn::make('name')
            ->requiredMapping()
            ->rules(['required', 'max:255']),
        ImportColumn::make('sku')
            ->label('SKU')
            ->requiredMapping()
            ->rules(['required', 'max:32']),
        ImportColumn::make('price')
            ->numeric()
            ->rules(['numeric', 'min:0']),
    ];
}
```

Customizing the label of an import column

The label for each column will be generated automatically from its name, but you can override it by calling the `label()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
    ->label('SKU')
```

Requiring an importer column to be mapped to a CSV column

You can call the `requiredMapping()` method to make a column required to be mapped to a column in the CSV. Columns that are required in the database should be required to be mapped:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
    ->requiredMapping()
```

If you require a column in the database, you also need to make sure that it has a `rules(['required'])` validation rule.

Validating CSV data

You can call the `rules()` method to add validation rules to a column. These rules will check the data in each row from the CSV before it is saved to the database:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
    ->rules(['required', 'max:32'])
```

Any rows that do not pass validation will not be imported. Instead, they will be compiled into a new CSV of "failed rows", which the user can download after the import has finished. The user will be shown a list of validation errors for each row that failed.

Casting state

Before [validation](#), data from the CSV can be cast. This is useful for converting strings into the correct data type, otherwise validation may fail. For example, if you have a `price` column in your CSV, you may want to cast it to a float:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('price')
    ->castStateUsing(function (string $state): ?float {
        if ($state === '') {
            return null;
        }

        $state = preg_replace('/[^0-9.]/', '', $state);
        $state = floatval($state);

        return round($state, precision: 2);
    })
}
```

In this example, we pass in a function that is used to cast the `$state`. This function removes any non-numeric characters from the string, casts it to a float, and rounds it to two decimal places.

Please note: if a column is not [required by validation](#), and it is empty, it will not be cast.

Filament also ships with some built-in casting methods:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('price')
    ->numeric() // Casts the state to a float.

ImportColumn::make('price')
    ->numeric(decimalPlaces: 2) // Casts the state to a float, and rounds it to 2 decimal
places.

ImportColumn::make('quantity')
    ->integer() // Casts the state to an integer.

ImportColumn::make('is_visible')
    ->boolean() // Casts the state to a boolean.
```

Mutating the state after it has been cast

If you're using a [built-in casting method](#) or [array cast](#), you can mutate the state after it has been cast by passing a function to the `castStateUsing()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('price')
->numeric()
->castStateUsing(function (float $state): ?float {
    if ($state == null) {
        return null;
    }

    return round($state * 100);
})
```

You can even access the original state before it was cast, by defining an `$originalState` argument in the function:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('price')
->numeric()
->castStateUsing(function (float $state, mixed $originalState): ?float {
    // ...
})
```

Importing relationships

You may use the `relationship()` method to import a relationship. At the moment, only `BelongsTo` relationships are supported. For example, if you have a `category` column in your CSV, you may want to import the category relationship:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('author')
->relationship()
```

In this example, the `author` column in the CSV will be mapped to the `author_id` column in the database. The CSV should contain the primary keys of authors, usually `id`.

If the column has a value, but the author cannot be found, the import will fail validation. Filament automatically adds validation to all relationship columns, to ensure that the relationship is not empty when it is required.

Customizing the relationship import resolution

If you want to find a related record using a different column, you can pass the column name as `resolveUsing`:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('author')
->relationship(resolveUsing: 'email')
```

You can pass in multiple columns to `resolveUsing`, and they will be used to find the author, in an "or" fashion. For example, if you pass in `['email', 'username']`, the record can be found by either their email or username:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('author')
    ->relationship(resolveUsing: ['email', 'username'])
```

You can also customize the resolution process, by passing in a function to `resolveUsing`, which should return a record to associate with the relationship:

```
use App\Models\Author;
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('author')
    ->relationship(resolveUsing: function (string $state): ?Author {
        return Author::query()
            ->where('email', $state)
            ->orWhere('username', $state)
            ->first();
    })
```

You could even use this function to dynamically determine which columns to use to resolve the record:

```
use App\Models\Author;
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('author')
    ->relationship(resolveUsing: function (string $state): ?Author {
        if (filter_var($state, FILTER_VALIDATE_EMAIL)) {
            return 'email';
        }

        return 'username';
    })
```

Handling multiple values in a single column as an array

You may use the `array()` method to cast the values in a column to an array. It accepts a delimiter as its first argument, which is used to split the values in the column into an array. For example, if you have a `documentation_urls` column in your CSV, you may want to cast it to an array of URLs:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('documentation_urls')
    ->array(',')
```

In this example, we pass in a comma as the delimiter, so the values in the column will be split by commas, and cast to an array.

Casting each item in an array

If you want to cast each item in the array to a different data type, you can chain the [built-in casting methods](#):

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('customer_ratings')
    ->array(',')
    ->integer() // Casts each item in the array to an integer.
```

Validating each item in an array

If you want to validate each item in the array, you can chain the `nestedRecursiveRules()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('customer_ratings')
    ->array(',')
    ->integer()
    ->rules(['array'])
    ->nestedRecursiveRules(['integer', 'min:1', 'max:5'])
```

Customizing how a column is filled into a record

If you want to customize how column state is filled into a record, you can pass a function to the `fillRecordUsing()` method:

```
use App\Models\Product;

ImportColumn::make('sku')
    ->fillRecordUsing(function (Product $record, string $state): void {
        $record->sku = strtoupper($state);
    })
```

Adding helper text below the import column

Sometimes, you may wish to provide extra information for the user before validation. You can do this by adding `helperText()` to a column, which gets displayed below the mapping select:

```
use Filament\Forms\Components\TextInput;

ImportColumn::make('skus')
    ->array(',')
    ->helperText('A comma-separated list of SKUs.')
```

Updating existing records when importing

When generating an importer class, you will see this `resolveRecord()` method:

```
use App\Models\Product;

public function resolveRecord(): ?Product
{
    // return Product::firstOrNew([
    //     // Update existing records, matching them by `$this->data['column_name']` 
    //     // 'email' => $this->data['email'],
    // ]);

    return new Product();
}
```

This method is called for each row in the CSV, and is responsible for returning a model instance that will be filled with the data from the CSV, and saved to the database. By default, it will create a new record for each row. However, you can customize this behavior to update existing records instead. For example, you might want to update a product if it already exists, and create a new one if it doesn't. To do this, you can uncomment the `firstOrNew()` line, and pass the column name that you want to match on. For a product, we might want to match on the `sku` column:

```
use App\Models\Product;

public function resolveRecord(): ?Product
{
    return Product::firstOrNew([
        'sku' => $this->data['sku'],
    ]);
}
```

Updating existing records when importing only

If you want to write an importer that only updates existing records, and does not create new ones, you can return `null` if no record is found:

```
use App\Models\Product;

public function resolveRecord(): ?Product
{
    return Product::query()
        ->where('sku', $this->data['sku'])
        ->first();
}
```

If you'd like to fail the import row if no record is found, you can throw a `RowImportFailedException` with a message:

```

use App\Models\Product;
use Filament\Actions\Imports\Exceptions\RowImportFailedException;

public function resolveRecord(): ?Product
{
    $product = Product::query()
        ->where('sku', $this->data['sku'])
        ->first();

    if (! $product) {
        throw new RowImportFailedException("No product found with SKU [{$this->data['sku']}].");
    }

    return $product;
}

```

When the import is completed, the user will be able to download a CSV of failed rows, which will contain the error messages.

Ignoring blank state for an import column

By default, if a column in the CSV is blank, and mapped by the user, and it's not required by validation, the column will be imported as `null` in the database. If you'd like to ignore blank state, and use the existing value in the database instead, you can call the `ignoreBlankState()` method:

```

use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('price')
    ->ignoreBlankState()

```

Using import options

The import action can render extra form components that the user can interact with when importing a CSV. This can be useful to allow the user to customize the behavior of the importer. For instance, you might want a user to be able to choose whether to update existing records when importing, or only create new ones. To do this, you can return options form components from the `getOptionsFormComponents()` method on your importer class:

```

use Filament\Forms\Components\Checkbox;

public static function getOptionsFormComponents(): array
{
    return [
        Checkbox::make('updateExisting')
            ->label('Update existing records'),
    ];
}

```

Alternatively, you can pass a set of static options to the importer through the `options()` method on the action:

```
use Filament\Actions\ImportAction;

ImportAction::make()
    ->importer(ProductImporter::class)
    ->options([
        'updateExisting' => true,
    ])
)
```

Now, you can access the data from these options inside the importer class, by calling `$this->options`. For example, you might want to use it inside `resolveRecord()` to update an existing product:

```
use App\Models\Product;

public function resolveRecord(): ?Product
{
    if ($this->options['updateExisting'] ?? false) {
        return Product::firstOrNew([
            'sku' => $this->data['sku'],
        ]);
    }

    return new Product();
}
```

Improving import column mapping guesses

By default, Filament will attempt to "guess" which columns in the CSV match which columns in the database, to save the user time. It does this by attempting to find different combinations of the column name, with spaces, `-`, , all cases insensitively. However, if you'd like to improve the guesses, you can call the `guess()` method with more examples of the column name that could be present in the CSV:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
    ->guess(['id', 'number', 'stock-keeping unit'])
```

Providing example CSV data

Before the user uploads a CSV, they have an option to download an example CSV file, containing all the available columns that can be imported. This is useful, as it allows the user to import this file directly into their spreadsheet software, and fill it out.

You can also add an example row to the CSV, to show the user what the data should look like. To fill in this example row, you can pass in an example column value to the `example()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
    ->example('ABC123')
```

Or if you want to add more than one example row, you can pass an array to the `examples()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
->examples(['ABC123', 'DEF456'])
```

By default, the name of the column is used in the header of the example CSV. You can customize the header per-column using `exampleHeader()`:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('sku')
->exampleHeader('SKU')
```

Using a custom user model

By default, the `imports` table has a `user_id` column. That column is constrained to the `users` table:

```
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
```

In the `Import` model, the `user()` relationship is defined as a `BelongsTo` relationship to the `App\Models\User` model. If the `App\Models\User` model does not exist, or you want to use a different one, you can bind a new `Authenticatable` model to the container in a service provider's `register()` method:

```
use App\Models\Admin;
use Illuminate\Contracts\Auth\Authenticatable;

$this->app->bind(Authenticatable::class, Admin::class);
```

If your authenticatable model uses a different table to `users`, you should pass that table name to `constrained()`:

```
$table->foreignId('user_id')->constrained('admins')->cascadeOnDelete();
```

Using a polymorphic user relationship

If you want to associate imports with multiple user models, you can use a polymorphic `MorphTo` relationship instead. To do this, you need to replace the `user_id` column in the `imports` table:

```
$table->morphs('user');
```

Then, in a service provider's `boot()` method, you should call `Import::polymorphicUserRelationship()` to swap the `user()` relationship on the `Import` model to a `MorphTo` relationship:

```
use Filament\Actions\Imports\Models\Import;

Import::polymorphicUserRelationship();
```

Limiting the maximum number of rows that can be imported

To prevent server overload, you may wish to limit the maximum number of rows that can be imported from one CSV file. You can do this by calling the `maxRows()` method on the action:

```
ImportAction::make()
->importer(ProductImporter::class)
->maxRows(100000)
```

Changing the import chunk size

Filament will chunk the CSV, and process each chunk in a different queued job. By default, chunks are 100 rows at a time. You can change this by calling the `chunkSize()` method on the action:

```
ImportAction::make()
->importer(ProductImporter::class)
->chunkSize(250)
```

If you are encountering memory or timeout issues when importing large CSV files, you may wish to reduce the chunk size.

Changing the CSV delimiter

The default delimiter for CSVs is the comma (,), If your import uses a different delimiter, you may call the `csvDelimiter()` method on the action, passing a new one:

```
ImportAction::make()
->importer(ProductImporter::class)
->csvDelimiter(';',')
```

You can only specify a single character, otherwise an exception will be thrown.

Changing the column header offset

If your column headers are not on the first row of the CSV, you can call the `headerOffset()` method on the action, passing the number of rows to skip:

```
ImportAction::make()
->importer(ProductImporter::class)
->headerOffset(5)
```

Customizing the import job

The default job for processing imports is `Filament\Actions\Imports\Jobs\ImportCsv`. If you want to extend this class and override any of its methods, you may replace the original class in the `register()` method of a service provider:

```
use App\Jobs\ImportCsv;
use Filament\Actions\Imports\Jobs\ImportCsv as BaseImportCsv;

$this->app->bind(BaseImportCsv::class, ImportCsv::class);
```

Or, you can pass the new job class to the `job()` method on the action, to customize the job for a specific import:

```
use App\Jobs\ImportCsv;

ImportAction::make()
    ->importer(ProductImporter::class)
    ->job(ImportCsv::class)
```

Customizing the import queue and connection

By default, the import system will use the default queue and connection. If you'd like to customize the queue used for jobs of a certain importer, you may override the `getJobQueue()` method in your importer class:

```
public function getJobQueue(): ?string
{
    return 'imports';
}
```

You can also customize the connection used for jobs of a certain importer, by overriding the `getJobConnection()` method in your importer class:

```
public function getJobConnection(): ?string
{
    return 'sqS';
}
```

Customizing the import job middleware

By default, the import system will only process one job at a time from each import. This is to prevent the server from being overloaded, and other jobs from being delayed by large imports. That functionality is defined in the `WithoutOverlapping` middleware on the importer class:

```
public function getJobMiddleware(): array
{
    return [
        new WithoutOverlapping("import{$this->import->getKey()}")->expireAfter(600),
    ];
}
```

If you'd like to customize the middleware that is applied to jobs of a certain importer, you may override this method in your importer class. You can read more about job middleware in the [Laravel docs](#).

Customizing the import job retries

By default, the import system will retry a job for 24 hours. This is to allow for temporary issues, such as the database being unavailable, to be resolved. That functionality is defined in the `getJobRetryUntil()` method on the importer class:

```
use Carbon\CarbonInterface;

public function getJobRetryUntil(): ?CarbonInterface
{
    return now()->addDay();
}
```

If you'd like to customize the retry time for jobs of a certain importer, you may override this method in your importer class. You can read more about job retries in the [Laravel docs](#).

Customizing the import job tags

By default, the import system will tag each job with the ID of the import. This is to allow you to easily find all jobs related to a certain import. That functionality is defined in the `getJobTags()` method on the importer class:

```
public function getJobTags(): array
{
    return ["import{$this->import->getKey()}"];
}
```

If you'd like to customize the tags that are applied to jobs of a certain importer, you may override this method in your importer class.

Customizing the import job batch name

By default, the import system doesn't define any name for the job batches. If you'd like to customize the name that is applied to job batches of a certain importer, you may override the `getJobBatchName()` method in your importer class:

```
public function getJobBatchName(): ?string
{
    return 'product-import';
}
```

Customizing import validation messages

The import system will automatically validate the CSV file before it is imported. If there are any errors, the user will be shown a list of them, and the import will not be processed. If you'd like to override any default validation messages, you may do so by overriding the `getValidationMessages()` method on your importer class:

```
public function getValidationMessages(): array
{
    return [
        'name.required' => 'The name column must not be empty.',
    ];
}
```

To learn more about customizing validation messages, read the [Laravel docs](#).

Customizing import validation attributes

When columns fail validation, their label is used in the error message. To customize the label used in field error messages, use the `validationAttribute()` method:

```
use Filament\Actions\Imports\ImportColumn;

ImportColumn::make('name')
    ->validationAttribute('full name')
```

Customizing import file validation

You can add new [Laravel validation rules](#) for the import file using the `fileRules()` method:

```
use Illuminate\Validation\Rules\File;

ImportAction::make()
    ->importer(ProductImporter::class)
    ->fileRules([
        'max:1024',
        // or
        File::types(['csv', 'txt'])->max(1024),
    ]),
]
```

Lifecycle hooks

Hooks may be used to execute code at various points within an importer's lifecycle, like before a record is saved. To set up a hook, create a protected method on the importer class with the name of the hook:

```
protected function beforeSave(): void
{
    // ...
}
```

In this example, the code in the `beforeSave()` method will be called before the validated data from the CSV is saved to the database.

There are several available hooks for importers:

```

use Filament\Actions\Imports\Importer;

class ProductImporter extends Importer
{
    // ...

    protected function beforeValidate(): void
    {
        // Runs before the CSV data for a row is validated.
    }

    protected function afterValidate(): void
    {
        // Runs after the CSV data for a row is validated.
    }

    protected function beforeFill(): void
    {
        // Runs before the validated CSV data for a row is filled into a model instance.
    }

    protected function afterFill(): void
    {
        // Runs after the validated CSV data for a row is filled into a model instance.
    }

    protected function beforeSave(): void
    {
        // Runs before a record is saved to the database.
    }

    protected function beforeCreate(): void
    {
        // Similar to `beforeSave()`, but only runs when creating a new record.
    }

    protected function beforeUpdate(): void
    {
        // Similar to `beforeSave()`, but only runs when updating an existing record.
    }

    protected function afterSave(): void
    {
        // Runs after a record is saved to the database.
    }

    protected function afterCreate(): void
    {
        // Similar to `afterSave()`, but only runs when creating a new record.
    }

    protected function afterUpdate(): void
    {
        // Similar to `afterSave()`, but only runs when updating an existing record.
    }
}

```

Inside these hooks, you can access the current row's data using `$this->data`. You can also access the original row of data from the CSV, before it was cast or mapped, using `$this->originalData`.

The current record (if it exists yet) is accessible in `$this->record`, and the import form options using `$this->options`.

Authorization

By default, only the user who started the import may access the failure CSV file that gets generated if part of an import fails. If you'd like to customize the authorization logic, you may create an `ImportPolicy` class, and register it in your AuthServiceProvider:

```
use App\Policies\ImportPolicy;
use Filament\Actions\Imports\Models\Import;

protected $policies = [
    Import::class => ImportPolicy::class,
];
```

The `view()` method of the policy will be used to authorize access to the failure CSV file.

Please note that if you define a policy, the existing logic of ensuring only the user who started the import can access the failure CSV file will be removed. You will need to add that logic to your policy if you want to keep it:

```
use App\Models\User;
use Filament\Actions\Imports\Models\Import;

public function view(User $user, Import $import): bool
{
    return $import->user()->is($user);
}
```

Export

Overview

Filament v3.2 introduced a prebuilt action that is able to export rows to a CSV or XLSX file. When the trigger button is clicked, a modal asks for the columns that they want to export, and what they should be labeled. This feature uses [job batches](#) and [database notifications](#), so you need to publish those migrations from Laravel. Also, you need to publish the migrations for tables that Filament uses to store information about exports:

```
# Laravel 11 and higher
php artisan make:queue-batches-table
php artisan make:notifications-table

# Laravel 10
php artisan queue:batches-table
php artisan notifications:table
```

```
# All apps
php artisan vendor:publish --tag=filament-actions-migrations
php artisan migrate
```

If you're using PostgreSQL, make sure that the `[data]` column in the notifications migration is using `[json()]`:
`$table->json('data');`.

If you're using UUIDs for your `[User]` model, make sure that your `[notifiable]` column in the notifications migration is using `[uuidMorphs()]`:
`$table->uuidMorphs('notifiable');`.

You may use the `[ExportAction]` like so:

```
use App\Filament\Exports\ProductExporter;
use Filament\Actions\ExportAction;

ExportAction::make()
    ->exporter(ProductExporter::class)
```

If you want to add this action to the header of a table instead, you can use `[Filament\Tables\Actions\ExportAction]`:

```
use App\Filament\Exports\ProductExporter;
use Filament\Tables\Actions\ExportAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->headerActions([
            ExportAction::make()
                ->exporter(ProductExporter::class)
        ]);
}
```

Or if you want to add it as a table bulk action, so that the user can choose which rows to export, they can use

`Filament\Tables\Actions\ExportBulkAction`:

```
use App\Filament\Exports\ProductExporter;
use Filament\Tables\Actions\ExportBulkAction;
use Filament\Tables\Table;

public function table(Table $table): Table
{
    return $table
        ->bulkActions([
            ExportBulkAction::make()
                ->exporter(ProductExporter::class)
        ]);
}
```

The “exporter” class needs to be created to tell Filament how to export each row.

Creating an exporter

To create an exporter class for a model, you may use the `make:filament-exporter` command, passing the name of a model:

```
php artisan make:filament-exporter Product
```

This will create a new class in the `app/Filament/Exports` directory. You now need to define the columns that can be exported.

Automatically generating exporter columns

If you'd like to save time, Filament can automatically generate the columns for you, based on your model's database columns, using `--generate`:

```
php artisan make:filament-exporter Product --generate
```

Defining exporter columns

To define the columns that can be exported, you need to override the `getColumns()` method on your exporter class, returning an array of `ExportColumn` objects:

```
use Filament\Actions\Exports\ExportColumn;

public static function getColumns(): array
{
    return [
        ExportColumn::make('name'),
        ExportColumn::make('sku')
            ->label('SKU'),
        ExportColumn::make('price'),
    ];
}
```

Customizing the label of an export column

The label for each column will be generated automatically from its name, but you can override it by calling the `label()` method:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('sku')
    ->label('SKU')
```

Configuring the default column selection

By default, all columns will be selected when the user is asked which columns they would like to export. You can customize the default selection state for a column with the `enabledByDefault()` method:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('description')
    ->enabledByDefault(false)
```

Disabling column selection

By default, user will be asked which columns they would like to export. You can disable this functionality using `columnMapping(false)`:

```
use App\Filament\Exports\ProductExporter;

ExportAction::make()
    ->exporter(ProductExporter::class)
    ->columnMapping(false)
```

Calculated export column state

Sometimes you need to calculate the state of a column, instead of directly reading it from a database column.

By passing a callback function to the `state()` method, you can customize the returned state for that column based on the `$record`:

```
use App\Models\Order;
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('amount_including_vat')
    ->state(function (Order $record): float {
        return $record->amount * (1 + $record->vat_rate);
    })
```

Formatting the value of an export column

You may instead pass a custom formatting callback to `formatStateUsing()`, which accepts the `$state` of the cell, and optionally the Eloquent `$record`:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('status')
    ->formatStateUsing(fn (string $state): string => __("statuses.{$state}"))
```

If there are multiple values in the column, the function will be called for each value.

Limiting text length

You may `limit()` the length of the cell's value:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('description')
    ->limit(50)
```

Limiting word count

You may limit the number of `words()` displayed in the cell:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('description')
    ->words(10)
```

Adding a prefix or suffix

You may add a `prefix()` or `suffix()` to the cell's value:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('domain')
    ->prefix('https://')
    ->suffix('.com')
```

Exporting multiple values in a cell

By default, if there are multiple values in the column, they will be comma-separated. You may use the `listAsJson()` method to list them as a JSON array instead:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('tags')
    ->listAsJson()
```

Displaying data from relationships

You may use "dot notation" to access columns within relationships. The name of the relationship comes first, followed by a period, followed by the name of the column to display:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('author.name')
```

Counting relationships

If you wish to count the number of related records in a column, you may use the `counts()` method:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('users_count')->counts('users')
```

In this example, `users` is the name of the relationship to count from. The name of the column must be `users_count`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Actions\Exports\ExportColumn;
use Illuminate\Database\Eloquent\Builder;

ExportColumn::make('users_count')->counts([
    'users' => fn (Builder $query) => $query->where('is_active', true),
])
```

Determining relationship existence

If you simply wish to indicate whether related records exist in a column, you may use the `exists()` method:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('users_exists')->exists('users')
```

In this example, `users` is the name of the relationship to check for existence. The name of the column must be `users_exists`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Actions\Exports\ExportColumn;
use Illuminate\Database\Eloquent\Builder;

ExportColumn::make('users_exists')->exists([
    'users' => fn (Builder $query) => $query->where('is_active', true),
])
```

Aggregating relationships

Filament provides several methods for aggregating a relationship field, including `avg()`, `max()`, `min()` and `sum()`. For instance, if you wish to show the average of a field on all related records in a column, you may use the `avg()` method:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('users_avg_age')->avg('users', 'age')
```

In this example, `users` is the name of the relationship, while `age` is the field that is being averaged. The name of the column must be `users_avg_age`, as this is the convention that [Laravel uses](#) for storing the result.

If you'd like to scope the relationship before calculating, you can pass an array to the method, where the key is the relationship name and the value is the function to scope the Eloquent query with:

```
use Filament\Actions\Exports\ExportColumn;
use Illuminate\Database\Eloquent\Builder;

ExportColumn::make('users_avg_age')->avg([
    'users' => fn (Builder $query) => $query->where('is_active', true),
], 'age')
```

Configuring the export formats

By default, the export action will allow the user to choose between both CSV and XLSX formats. You can use the `ExportFormat` enum to customize this, by passing an array of formats to the `formats()` method on the action:

```
use App\Filament\Exports\ProductExporter;
use Filament\Actions\Exports\Enums\ExportFormat;

ExportAction::make()
    ->exporter(ProductExporter::class)
    ->formats([
        ExportFormat::Csv,
    ])
    // or
    ->formats([
        ExportFormat::Xlsx,
    ])
    // or
    ->formats([
        ExportFormat::Xlsx,
        ExportFormat::Csv,
    ])
)
```

Alternatively, you can override the `getFormats()` method on the exporter class, which will set the default formats for all actions that use that exporter:

```
use Filament\Actions\Exports\Enums\ExportFormat;

public function getFormats(): array
{
    return [
        ExportFormat::Csv,
    ];
}
```

Modifying the export query

By default, if you are using the `ExportAction` with a table, the action will use the table's currently filtered and sorted query to export the data. If you don't have a table, it will use the model's default query. To modify the query builder before exporting, you can use the `modifyQueryUsing()` method on the action:

```
use App\Filament\Exports\ProductExporter;
use Illuminate\Database\Eloquent\Builder;

ExportAction::make()
->exporter(ProductExporter::class)
->modifyQueryUsing(fn (Builder $query) => $query->where('is_active', true))
```

Alternatively, you can override the `modifyQuery()` method on the exporter class, which will modify the query for all actions that use that exporter:

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Relations\MorphTo;

public static function modifyQuery(Builder $query): Builder
{
    return $query->with([
        'purchasable' => fn (MorphTo $morphTo) => $morphTo->morphWith([
            ProductPurchase::class => ['product'],
            ServicePurchase::class => ['service'],
            Subscription::class => ['plan'],
        ]),
    ]);
}
```

Configuring the export filesystem

Customizing the storage disk

By default, exported files will be uploaded to the storage disk defined in the [configuration file](#). You can also set the `FILAMENT_FILESYSTEM_DISK` environment variable to change this.

If you want to use a different disk for a specific export, you can pass the disk name to the `disk()` method on the action:

```
ExportAction::make()
->exporter(ProductExporter::class)
->fileDisk('s3')
```

Alternatively, you can override the `getFileDisk()` method on the exporter class, returning the name of the disk:

```
public function getFileDisk(): string
{
    return 's3';
}
```

Configuring the export file names

By default, exported files will have a name generated based on the ID and type of the export. You can also use the `fileName()` method on the action to customize the file name:

```
use Filament\Actions\Exports\Models\Export;

ExportAction::make()
    ->exporter(ProductExporter::class)
    ->fileName(fn (Export $export): string => "products-{$export->getKey()}.csv")
```

Alternatively, you can override the `getFileName()` method on the exporter class, returning a string:

```
use Filament\Actions\Exports\Models\Export;

public function getFileName(Export $export): string
{
    return "products-{$export->getKey()}.csv";
}
```

Using export options

The export action can render extra form components that the user can interact with when exporting a CSV. This can be useful to allow the user to customize the behavior of the exporter. For instance, you might want a user to be able to choose the format of specific columns when exporting. To do this, you can return options form components from the `getOptionsFormComponents()` method on your exporter class:

```
use Filament\Forms\Components\TextInput;

public static function getOptionsFormComponents(): array
{
    return [
        TextInput::make('descriptionLimit')
            ->label('Limit the length of the description column content')
            ->integer(),
    ];
}
```

Alternatively, you can pass a set of static options to the exporter through the `options()` method on the action:

```
ExportAction::make()
    ->exporter(ProductExporter::class)
    ->options([
        'descriptionLimit' => 250,
    ])
```

Now, you can access the data from these options inside the exporter class, by injecting the `[$options]` argument into any closure function. For example, you might want to use it inside `formatStateUsing()` to format a column's value:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('description')
    ->formatStateUsing(function (string $state, array $options): string {
        return (string) str($state)->limit($options['descriptionLimit'] ?? 100);
    })
```

Alternatively, since the `$options` argument is passed to all closure functions, you can access it inside `limit()`:

```
use Filament\Actions\Exports\ExportColumn;

ExportColumn::make('description')
    ->limit(fn (array $options): int => $options['descriptionLimit'] ?? 100)
```

Using a custom user model

By default, the `exports` table has a `user_id` column. That column is constrained to the `users` table:

```
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
```

In the `Export` model, the `user()` relationship is defined as a `BelongsTo` relationship to the `App\Models\User` model. If the `App\Models\User` model does not exist, or you want to use a different one, you can bind a new `Authenticatable` model to the container in a service provider's `register()` method:

```
use App\Models\Admin;
use Illuminate\Contracts\Auth\Authenticatable;

$this->app->bind(Authenticatable::class, Admin::class);
```

If your authenticatable model uses a different table to `users`, you should pass that table name to `constrained()`:

```
$table->foreignId('user_id')->constrained('admins')->cascadeOnDelete();
```

Using a polymorphic user relationship

If you want to associate exports with multiple user models, you can use a polymorphic `MorphTo` relationship instead. To do this, you need to replace the `user_id` column in the `exports` table:

```
$table->morphs('user');
```

Then, in a service provider's `boot()` method, you should call `Export::polymorphicUserRelationship()` to swap the `user()` relationship on the `Export` model to a `MorphTo` relationship:

```
use Filament\Actions\Exports\Models\Export;

Export::polymorphicUserRelationship();
```

Limiting the maximum number of rows that can be exported

To prevent server overload, you may wish to limit the maximum number of rows that can be exported from one CSV file. You can do this by calling the `maxRows()` method on the action:

```
ExportAction::make()
    ->exporter(ProductExporter::class)
    ->maxRows(100000)
```

Changing the export chunk size

Filament will chunk the CSV, and process each chunk in a different queued job. By default, chunks are 100 rows at a time. You can change this by calling the `chunkSize()` method on the action:

```
ExportAction::make()
    ->exporter(ProductExporter::class)
    ->chunkSize(250)
```

If you are encountering memory or timeout issues when exporting large CSV files, you may wish to reduce the chunk size.

Changing the CSV delimiter

The default delimiter for CSVs is the comma (,), If you want to export using a different delimiter, you may override the `getCsvDelimiter()` method on the exporter class, returning a new one:

```
public static function getCsvDelimiter(): string
{
    return ';';
}
```

You can only specify a single character, otherwise an exception will be thrown.

Styling XLSX cells

If you want to style the cells of the XLSX file, you may override the `getXlsxCellStyle()` method on the exporter class, returning an [OpenSpout Style object](#):

```
use OpenSpout\Common\Entity\Style\Style;

public function getXlsxCellStyle(): ?Style
{
    return (new Style())
        ->setFontSize(12)
        ->setFontName('Consolas');
}
```

If you want to use a different style for the header cells of the XLSX file only, you may override the `getXlsxHeaderCellStyle()` method on the exporter class, returning an [OpenSpout Style object](#):

```

use OpenSpout\Common\Entity\Style\CellAlignment;
use OpenSpout\Common\Entity\Style\CellVerticalAlignment;
use OpenSpout\Common\Entity\Style\Color;
use OpenSpout\Common\Entity\Style\Style;

public function getXlsxHeaderCellStyle(): ?Style
{
    return (new Style())
        ->setFontBold()
        ->setFontItalic()
        ->setFontSize(14)
        ->setFontName('Consolas')
        ->setFontColor(Color::rgb(255, 255, 77))
        ->setBackgroundColor(Color::rgb(0, 0, 0))
        ->setCellAlignment(CellAlignment::CENTER)
        ->setCellVerticalAlignment(CellVerticalAlignment::CENTER);
}

```

Customizing the export job

The default job for processing exports is `Filament\Actions\Exports\Jobs\PrepareCsvExport`. If you want to extend this class and override any of its methods, you may replace the original class in the `register()` method of a service provider:

```

use App\Jobs\PrepareCsvExport;
use Filament\Actions\Exports\Jobs\PrepareCsvExport as BasePrepareCsvExport;

$this->app->bind(BasePrepareCsvExport::class, PrepareCsvExport::class);

```

Or, you can pass the new job class to the `job()` method on the action, to customize the job for a specific export:

```

use App\Jobs\PrepareCsvExport;

ExportAction::make()
    ->exporter(ProductExporter::class)
    ->job(PrepareCsvExport::class)

```

Customizing the export queue and connection

By default, the export system will use the default queue and connection. If you'd like to customize the queue used for jobs of a certain exporter, you may override the `getJobQueue()` method in your exporter class:

```

public function getJobQueue(): ?string
{
    return 'exports';
}

```

You can also customize the connection used for jobs of a certain exporter, by overriding the `getJobConnection()` method in your exporter class:

```
public function getJobConnection(): ?string
{
    return 'sq';
}
```

Customizing the export job middleware

By default, the export system will only process one job at a time from each export. This is to prevent the server from being overloaded, and other jobs from being delayed by large exports. That functionality is defined in the `WithoutOverlapping` middleware on the exporter class:

```
public function getJobMiddleware(): array
{
    return [
        (new WithoutOverlapping("export{$this->export->getKey()}"))->expireAfter(600),
    ];
}
```

If you'd like to customize the middleware that is applied to jobs of a certain exporter, you may override this method in your exporter class. You can read more about job middleware in the [Laravel docs](#).

Customizing the export job retries

By default, the export system will retry a job for 24 hours. This is to allow for temporary issues, such as the database being unavailable, to be resolved. That functionality is defined in the `getJobRetryUntil()` method on the exporter class:

```
use Carbon\CarbonInterface;

public function getJobRetryUntil(): ?CarbonInterface
{
    return now()->addDay();
}
```

If you'd like to customize the retry time for jobs of a certain exporter, you may override this method in your exporter class. You can read more about job retries in the [Laravel docs](#).

Customizing the export job tags

By default, the export system will tag each job with the ID of the export. This is to allow you to easily find all jobs related to a certain export. That functionality is defined in the `getJobTags()` method on the exporter class:

```
public function getJobTags(): array
{
    return ["export{$this->export->getKey()}"];
}
```

If you'd like to customize the tags that are applied to jobs of a certain exporter, you may override this method in your exporter class.

Customizing the export job batch name

By default, the export system doesn't define any name for the job batches. If you'd like to customize the name that is applied to job batches of a certain exporter, you may override the `getJobBatchName()` method in your exporter class:

```
public function getJobBatchName(): ?string
{
    return 'product-export';
}
```

Authorization

By default, only the user who started the export may download files that get generated. If you'd like to customize the authorization logic, you may create an `ExportPolicy` class, and register it in your `AuthServiceProvider`:

```
use App\Policies\ExportPolicy;
use Filament\Actions\Exports\Models\Export;

protected $policies = [
    Export::class => ExportPolicy::class,
];
```

The `view()` method of the policy will be used to authorize access to the downloads.

Please note that if you define a policy, the existing logic of ensuring only the user who started the export can access it will be removed. You will need to add that logic to your policy if you want to keep it:

```
use App\Models\User;
use Filament\Actions\Exports\Models\Export;

public function view(User $user, Export $export): bool
{
    return $export->user()->is($user);
}
```

Advanced

Action utility injection

The vast majority of methods used to configure actions accept functions as parameters instead of hardcoded values:

```
Action::make('edit')
->label('Edit post')
->url(fn (): string => route('posts.edit', ['post' => $this->post]))
```

This alone unlocks many customization possibilities.

The package is also able to inject many utilities to use inside these functions, as parameters. All customization methods that accept functions as arguments can inject utilities.

These injected utilities require specific parameter names to be used. Otherwise, Filament doesn't know what to inject.

Injecting the current modal form data

If you wish to access the current modal form data, define a `[$data]` parameter:

```
function (array $data) {
    // ...
}
```

Be aware that this will be empty if the modal has not been submitted yet.

Injecting the current arguments

If you wish to access the current arguments that have been passed to the action, define an `[$arguments]` parameter:

```
function (array $arguments) {
    // ...
}
```

Injecting the current Livewire component instance

If you wish to access the current Livewire component instance that the action belongs to, define a `[$livewire]` parameter:

```
use Livewire\Component;

function (Component $livewire) {
    // ...
}
```

Injecting the current action instance

If you wish to access the current action instance, define a `[$action]` parameter:

```
function (Action $action) {
    // ...
}
```

Injecting multiple utilities

The parameters are injected dynamically using reflection, so you are able to combine multiple parameters in any order:

```
use Livewire\Component;

function (array $arguments, Component $livewire) {
    // ...
}
```

Injecting dependencies from Laravel's container

You may inject anything from Laravel's container like normal, alongside utilities:

```
use Illuminate\Http\Request;

function (Request $request, array $arguments) {
    // ...
}
```

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Since all actions are mounted to a Livewire component, we're just using Livewire testing helpers everywhere. If you've never tested Livewire components before, please read [this guide](#) from the Livewire docs.

Getting started

You can call an action by passing its name or class to `callAction()`:

```
use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callAction('send');

    expect($invoice->refresh())
        ->isSent()->toBeTrue();
});

});
```

To pass an array of data into an action, use the `data` parameter:

```
use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callAction('send', data: [
        'email' => $email = fake()->email(),
    ])
    ->assertHasNoActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($email);
});

});
```

If you ever need to only set an action's data without immediately calling it, you can use `setActionData()`:

```
use function Pest\Livewire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountAction('send')
    ->setActionData('send', data: [
        'email' => $email = fake()->email(),
    ])
});
});
```

Execution

To check if an action has been halted, you can use `assertActionHalted()`:

```
use function Pest\Livewire\livewire;

it('stops sending if invoice has no email address', function () {
    $invoice = Invoice::factory(['email' => null])->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callAction('send')
    ->assertActionHalted('send');
});
});
```

Errors

`assertHasNoActionErrors()` is used to assert that no validation errors occurred when submitting the action form.

To check if a validation error has occurred with the data, use `assertHasActionErrors()`, similar to `assertHasErrors()` in Livewire:

```
use function Pest\Livewire\livewire;

it('can validate invoice recipient email', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callAction('send', data: [
        'email' => Str::random(),
    ])
    ->assertHasActionErrors(['email' => ['email']]);
});
});
```

To check if an action is pre-filled with data, you can use the `assertActionDataSet()` method:

```
use function Pest\LiveWire\livewire;

it('can send invoices to the primary contact by default', function () {
    $invoice = Invoice::factory()->create();
    $recipientEmail = $invoice->company->primaryContact->email;

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountAction('send')
    ->assertActionDataSet([
        'email' => $recipientEmail,
    ])
    ->callMountedAction()
    ->assertHasNoActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($recipientEmail);
});
});
```

Action state

To ensure that an action exists or doesn't, you can use the `assertActionExists()` or `assertActionDoesNotExist()` method:

```
use function Pest\LiveWire\livewire;

it('can send but not unsend invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionExists('send')
    ->assertActionDoesNotExist('unsend');
});
```

To ensure an action is hidden or visible for a user, you can use the `assertActionHidden()` or `assertActionVisible()` methods:

```
use function Pest\LiveWire\livewire;

it('can only print invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionHidden('send')
    ->assertActionVisible('print');
});
```

To ensure an action is enabled or disabled for a user, you can use the `assertActionEnabled()` or `assertActionDisabled()` methods:

```
use function Pest\LiveWire\livewire;

it('can only print a sent invoice', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertActionDisabled('send')
        ->assertActionEnabled('print');
});
```

To ensure sets of actions exist in the correct order, you can use `assertActionsExistInOrder()`:

```
use function Pest\LiveWire\livewire;

it('can have actions in order', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertActionsExistInOrder(['send', 'export']);
});
```

To check if an action is hidden to a user, you can use the `assertActionHidden()` method:

```
use function Pest\LiveWire\livewire;

it('can not send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertActionHidden('send');
});
```

Button appearance

To ensure an action has the correct label, you can use `assertActionHasLabel()` and `assertActionDoesNotHaveLabel()`:

```
use function Pest\Livewire\livewire;

it('send action has correct label', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionHasLabel('send', 'Email Invoice')
    ->assertActionDoesNotHaveLabel('send', 'Send');
});
```

To ensure an action's button is showing the correct icon, you can use `assertActionHasIcon()` or `assertActionDoesNotHaveIcon()`:

```
use function Pest\Livewire\livewire;

it('when enabled the send button has correct icon', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionEnabled('send')
    ->assertActionHasIcon('send', 'envelope-open')
    ->assertActionDoesNotHaveIcon('send', 'envelope');
});
```

To ensure that an action's button is displaying the right color, you can use `assertActionHasColor()` or `assertActionDoesNotHaveColor()`:

```
use function Pest\Livewire\livewire;

it('actions display proper colors', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionHasColor('delete', 'danger')
    ->assertActionDoesNotHaveColor('print', 'danger');
});
```

URL

To ensure an action has the correct URL, you can use `assertActionHasUrl()`, `assertActionDoesNotHaveUrl()`, `assertActionShouldOpenUrlInNewTab()`, and `assertActionShouldNotOpenUrlInNewTab()`:

```
use function Pest\LiveWire\liveWire;

it('links to the correct Filament sites', function () {
    $invoice = Invoice::factory()->create();

    liveWire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertActionHasUrl('filament', 'https://filamentphp.com/')
    ->assertActionDoesNotHaveUrl('filament', 'https://github.com/filamentphp/filament')
    ->assertActionShouldOpenUrlInNewTab('filament')
    ->assertActionShouldNotOpenUrlInNewTab('github');
});
```

Upgrade Guide

If you see anything missing from this guide, please do not hesitate to [make a pull request](#) to our repository! Any help is appreciated!

New requirements

- Laravel v9.0+
- Livewire v3.0+

Please upgrade Filament before upgrading to Livewire v3. Instructions on how to upgrade Livewire can be found [here](#).

Upgrading automatically

The easiest way to upgrade your app is to run the automated upgrade script. This script will automatically upgrade your application to the latest version of Filament and make changes to your code, which handles most breaking changes.

```
composer require filament/upgrade:"^3.2" -W --dev
vendor/bin/filament-v3
```

Make sure to carefully follow the instructions, and review the changes made by the script. You may need to make some manual changes to your code afterwards, but the script should handle most of the repetitive work for you.

Finally, you must run `php artisan filament:install` to finalize the Filament v3 installation. This command must be run for all new Filament projects.

You can now `composer remove filament/upgrade` as you don't need it anymore.

Some plugins you're using may not be available in v3 just yet. You could temporarily remove them from your `composer.json` file until they've been upgraded, replace them with a similar plugins that are v3-compatible, wait for the plugins to be upgraded before upgrading your app, or even write PRs to help the authors upgrade them.

Upgrading manually

After upgrading the dependency via Composer, you should execute `php artisan filament:upgrade` in order to clear any Laravel caches and publish the new frontend assets.

Low-impact changes

Action execution with forms

In v2, if you passed a string to the `action()` function, and used a modal, the method with that name on the class would be run when the modal was submitted:

```
Action::make('import_data')
->action('importData')
->form([
    FileUpload::make('file'),
])
```

In v3, passing a string to the `action()` hard-wires it to run that action when the trigger button is clicked, so it does not open a modal.

It is recommended to place your logic in a closure function instead. See the [documentation](#) for more information.

One easy alternative is using `Closure::fromCallable()`:

```
Action::make('import_data')
->action(Closure::fromCallable([$this, 'importData']))
```

Chapter 6

Infolist Builder

Installation

The Infolist Builder package is pre-installed with the [Panel Builder](#). This guide is for using the Infolist Builder package in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

Require the Infolist Builder package using Composer:

```
composer require filament/infolists:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --infolists
npm install
npm run dev
```

Existing Laravel projects

Run the following command to install the Infolist Builder package assets:

```
php artisan filament:install --infolists
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament [preset](#) (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post-autoload-dump` hook, you can run the command yourself:

```
composer update
```

```
php artisan filament:upgrade
```

Getting Started

Overview

Filament's infolist package allows you to [render a read-only list of data about a particular entity](#). It's also used within other Filament packages, such as the [Panel Builder](#) for displaying [app resources](#) and [relation managers](#), as well as for [action modals](#). Learning the features of the Infolist Builder will be incredibly time-saving when both building your own custom Livewire applications and using Filament's other packages.

This guide will walk you through the basics of building infolists with Filament's infolist package. If you're planning to add a new infolist to your own Livewire component, you should [do that first](#) and then come back. If you're adding an infolist to an [app resource](#), or another Filament package, you're ready to go!

Defining entries

The first step to building an infolist is to define the entries that will be displayed in the list. You can do this by calling the `schema()` method on an [Infolist](#) object. This method accepts an array of entry objects.

```
use Filament\Infolists\Components\TextEntry;

$infolist
->schema([
    TextEntry::make('title'),
    TextEntry::make('slug'),
    TextEntry::make('content'),
]);
```

Each entry is a piece of information that should be displayed in the infolist. The `TextEntry` is used for displaying text, but there are [other entry types available](#).

Infolists within the Panel Builder and other packages usually have 2 columns by default. For custom infolists, you can use the `columns()` method to achieve the same effect:

```
$infolist
->schema([
    // ...
])
->columns(2);
```

Now, the `content` entry will only consume half of the available width. We can use the `columnSpan()` method to make it span the full width:

```
use Filament\Infolists\Components\TextEntry;

[
    TextEntry::make('title'),
    TextEntry::make('slug')
    TextEntry::make('content')
        ->columnSpan(2), // or `columnSpan('full')`,
]
```

You can learn more about columns and spans in the [layout documentation](#). You can even make them responsive!

Using layout components

The Infolist Builder allows you to use [layout components](#) inside the schema array to control how entries are displayed. `Section` is a layout component, and it allows you to add a heading and description to a set of entries. It can also allow entries inside it to collapse, which saves space in long infolists.

```
use Filament\Infolists\Components\Section;
use Filament\Infolists\Components\TextEntry;

[
    TextEntry::make('title'),
    TextEntry::make('slug'),
    TextEntry::make('content')
        ->columnSpan(2)
        ->markdown(),
    Section::make('Media')
        ->description('Images used in the page layout.')
        ->schema([
            // ...
        ]),
]
]
```

In this example, you can see how the `Section` component has its own `schema()` method. You can use this to nest other entries and layout components inside:

```
use Filament\Infolists\Components\ImageEntry;
use Filament\Infolists\Components\Section;
use Filament\Infolists\Components\TextEntry;

Section::make('Media')
    ->description('Images used in the page layout.')
    ->schema([
        ImageEntry::make('hero_image'),
        TextEntry::make('alt_text'),
    ])
]
```

This section now contains an `ImageEntry` and a `TextEntry`. You can learn more about those entries and their functionalities on the respective docs pages.

Next steps with the infolists package

Now you've finished reading this guide, where to next? Here are some suggestions:

- [Explore the available entries to display data in your infolist.](#)
- [Discover how to build complex, responsive layouts without touching CSS.](#)

Entries

Getting Started

Overview

Entry classes can be found in the `Filament\Infolists\Components` namespace. You can put them inside the `$infolist->schema()` method:

```
use Filament\Infolists\Infolist;

public function infolist(Infolist $infolist): Infolist
{
    return $infolist
        ->schema([
            // ...
        ]);
}
```

If you're inside a panel builder resource, the `infolist()` method should be static:

```
use Filament\Infolists\Infolist;

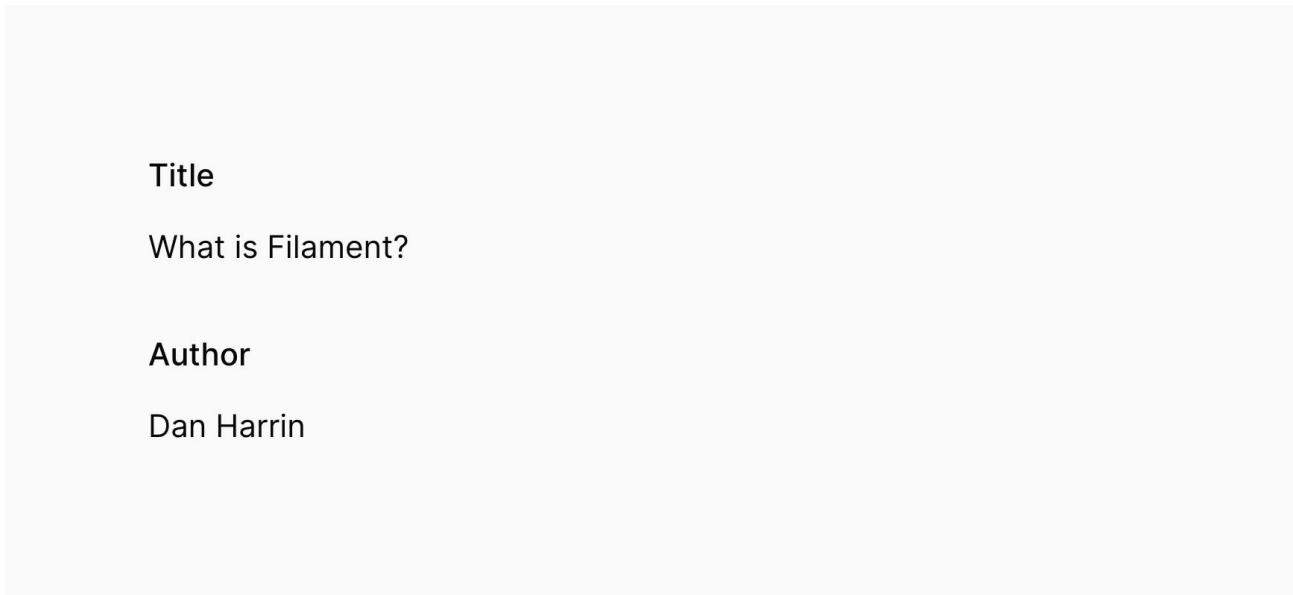
public static function infolist(Infolist $infolist): Infolist
{
    return $infolist
        ->schema([
            // ...
        ]);
}
```

Entries may be created using the static `make()` method, passing its unique name. You may use "dot notation" to access entries within relationships.

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')

TextEntry::make('author.name')
```



Available entries

- [Text entry](#)
- [Icon entry](#)
- [Image entry](#)
- [Color entry](#)
- [Key-value entry](#)
- [Repeatable entry](#)

You may also [create your own custom entries](#) to display data however you wish.

Setting a label

By default, the label of the entry, which is displayed in the header of the infolist, is generated from the name of the entry. You may customize this using the `label()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
    ->label('Post title')
```

Optionally, you can have the label automatically translated [using Laravel's localization features](#) with the `translateLabel()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
    ->translateLabel() // Equivalent to `label(__('Title'))`
```

Entry URLs

When an entry is clicked, you may open a URL.

Opening URLs

To open a URL, you may use the `url()` method, passing a callback or static URL to open. Callbacks accept a `$record` parameter which you may use to customize the URL:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
->url(fn (Post $record): string => route('posts.edit', ['post' => $record]))
```

You may also choose to open the URL in a new tab:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
->url(fn (Post $record): string => route('posts.edit', ['post' => $record]))
->openUrlInNewTab()
```

Setting a default value

To set a default value for entries with an empty state, you may use the `default()` method. This method will treat the default state as if it were real, so entries like `image` or `color` will display the default image or color.

```
use Filament\Infolists\Components\TextEntry;

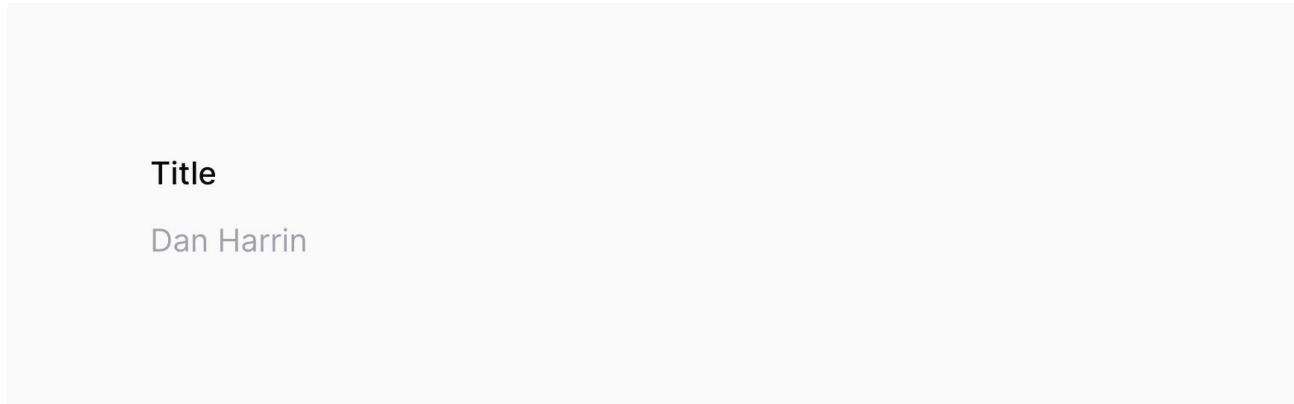
TextEntry::make('title')
->default('Untitled')
```

Adding placeholder text if an entry is empty

Sometimes you may want to display placeholder text for entries with an empty state, which is styled as a lighter gray text. This differs from the [default value](#), as the placeholder is always text and not treated as if it were real state.

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
->placeholder('Untitled')
```



Title

Dan Harrin

Adding helper text below the entry

Sometimes, you may wish to provide extra information for the user of the infolist. For this purpose, you may add helper text below the entry.

The `helperText()` method is used to add helper text:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('name')
->helperText('Your full name here, including any middle names.')
```

This method accepts a plain text string, or an instance of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even markdown, in the helper text:

```
use Filament\Infolists\Components\TextEntry;
use Illuminate\Support\HtmlString;

TextEntry::make('name')
->helperText(new HtmlString('Your <strong>full name</strong> here, including any middle
names.'))

TextEntry::make('name')
->helperText(str('Your **full name** here, including any middle names.')->inlineMarkdown()-
>toHtmlString())

TextEntry::make('name')
->helperText(view('name-helper-text'))
```

Name

Dan Harrin

Your **full name** here, including any middle names.

Adding a hint next to the label

As well as helper text below the entry, you may also add a "hint" next to the label of the entry. This is useful for displaying additional information about the entry, such as a link to a help page.

The `hint()` method is used to add a hint:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->hint('Documentation? What documentation?!')
```

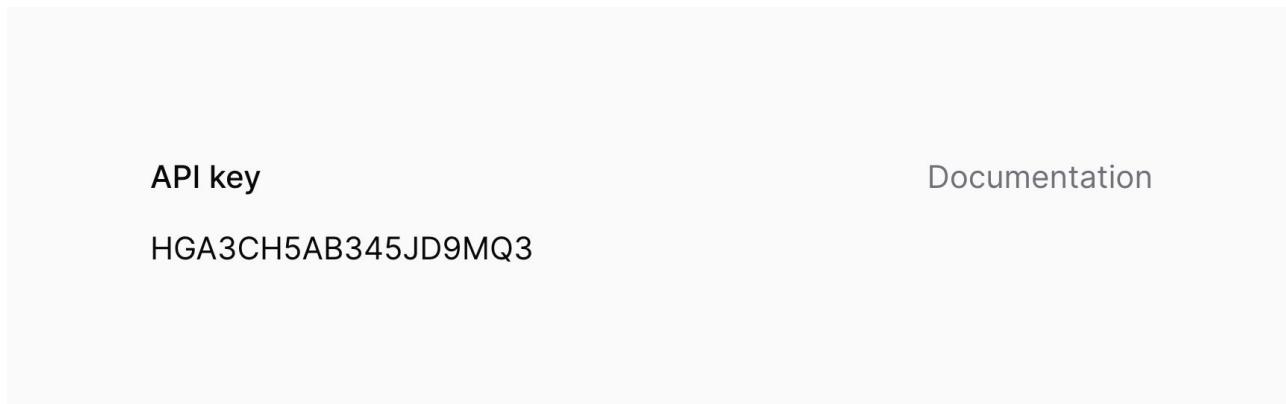
This method accepts a plain text string, or an instance of `Illuminate\Support\HtmlString` or `Illuminate\Contracts\Support\Htmlable`. This allows you to render HTML, or even markdown, in the helper text:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->hint(new HtmlString('<a href="/documentation">Documentation</a>'))

TextEntry::make('apiKey')
->label('API key')
->hint(str('[Documentation] (/documentation)')->inlineMarkdown()->toHtmlString())

TextEntry::make('apiKey')
->label('API key')
->hint(view('api-key-hint'))
```



The screenshot shows a filament input component. On the left, the label 'API key' is displayed above the input field. To the right of the input field, the hint 'Documentation' is shown in gray text. Below the input field, the value 'HGA3CH5AB345JD9MQ3' is displayed.

API key

Documentation

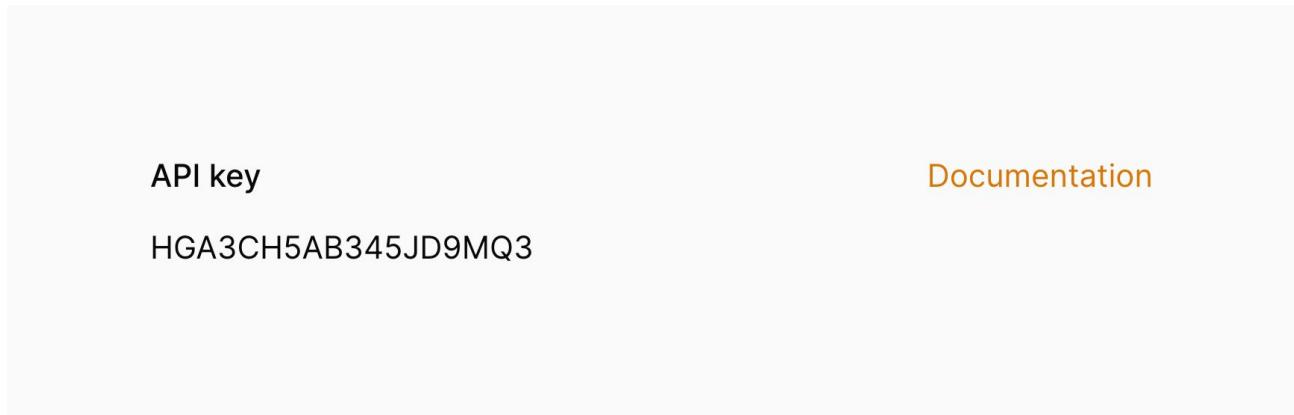
HGA3CH5AB345JD9MQ3

Changing the text color of the hint

You can change the text color of the hint. By default, it's gray, but you may use `danger`, `info`, `primary`, `success` and `warning`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->hint(str('[Documentation] (/documentation)')->inlineMarkdown()->toHtmlString())
->hintColor('primary')
```

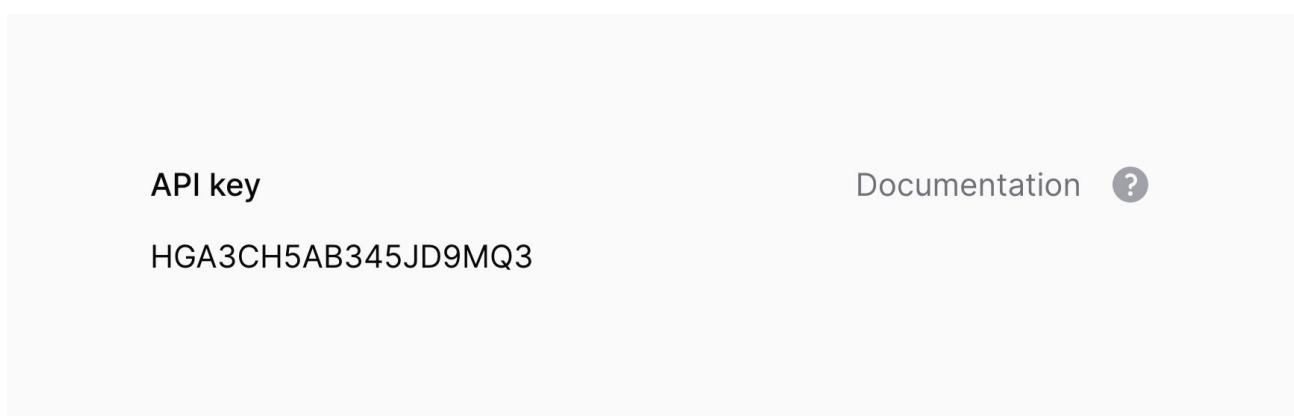


Adding an icon aside the hint

Hints may also have an `icon` rendered next to them:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->hint(str('[Documentation] (/documentation)')->inlineMarkdown() ->toHtmlString())
->hintIcon('heroicon-m-question-mark-circle')
```



Adding a tooltip to a hint icon

Additionally, you can add a tooltip to display when you hover over the hint icon, using the `tooltip` parameter of `hintIcon()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->hint(str('[Documentation] (/documentation)')->inlineMarkdown() ->toHtmlString())
->hintIcon('heroicon-m-question-mark-circle', tooltip: 'Read it!')
```

Hiding entries

To hide an entry conditionally, you may use the `hidden()` and `visible()` methods, whichever you prefer:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('role')
->hidden(! auth()->user()->isAdmin())
// or
TextEntry::make('role')
->visible(auth()->user()->isAdmin())
```

Calculated state

Sometimes you need to calculate the state of an entry, instead of directly reading it from a database entry.

By passing a callback function to the `state()` method, you can customize the returned state for that entry:

```
Infolists\Components\TextEntry::make('amount_including_vat')
->state(function (Model $record): float {
    return $record->amount * (1 + $record->vat_rate);
})
```

Tooltips

You may specify a tooltip to display when you hover over an entry:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
->tooltip('Shown at the top of the page')
```

Title

Shown at the top of the page

What is Filament?

This method also accepts a closure that can access the current infolist record:

```
use Filament\Infolists\Components\TextEntry;
use Illuminate\Database\Eloquent\Model;

TextEntry::make('title')
->tooltip(fn (Model $record): string => "By {$record->author->name}")
```

Custom attributes

The HTML of entries can be customized, by passing an array of `extraAttributes()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('slug')
->extraAttributes(['class' => 'bg-gray-200'])
```

These get merged onto the outer `<div>` element of each entry in that entry.

You can also pass extra HTML attributes to the entry wrapper which surrounds the label, entry, and any other text:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('slug')
->extraEntryWrapperAttributes(['class' => 'entry-locked'])
```

Global settings

If you wish to change the default behavior of all entries globally, then you can call the static `configureUsing()` method inside a service provider's `boot()` method, to which you pass a Closure to modify the entries using. For example, if you wish to make all `TextEntry` components `words(10)`, you can do it like so:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::configureUsing(function (TextEntry $entry): void {
    $entry
        ->words(10);
});
```

Of course, you are still able to overwrite this on each entry individually:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('name')
->words(null)
```

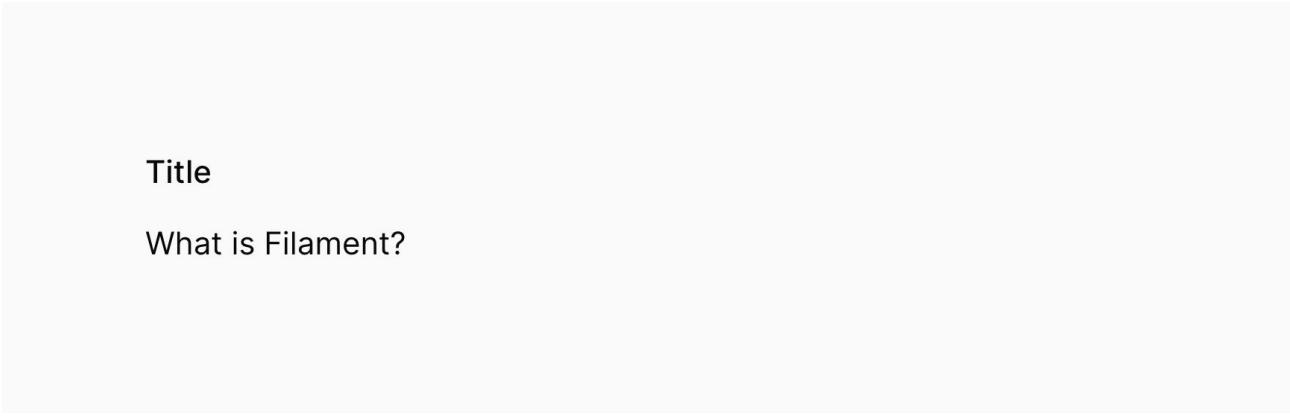
Text

Overview

Text entries display simple text:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
```



Title

What is Filament?

Displaying as a "badge"

By default, text is quite plain and has no background color. You can make it appear as a "badge" instead using the `badge()` method. A great use case for this is with statuses, where you may want to display a badge with a color that matches the status:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('status')
->badge()
->color(fn (string $state): string => match ($state) {
    'draft' => 'gray',
    'reviewing' => 'warning',
    'published' => 'success',
    'rejected' => 'danger',
})
```

Status

`published`

You may add other things to the badge, like an [icon](#).

Date formatting

You may use the `date()` and `dateTime()` methods to format the entry's state using [PHP date formatting tokens](#):

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('created_at')
    ->dateTime()
```

You may use the `since()` method to format the entry's state using [Carbon's diffForHumans\(\)](#):

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('created_at')
    ->since()
```

Additionally, you can use the `dateTooltip()`, `dateTimeTooltip()` or `timeTooltip()` method to display a formatted date in a tooltip, often to provide extra information:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('created_at')
    ->since()
    ->dateTimeTooltip()
```

Number formatting

The `numeric()` method allows you to format an entry as a number:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('stock')
    ->numeric()
```

If you would like to customize the number of decimal places used to format the number with, you can use the `decimalPlaces` argument:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('stock')
->numeric(decimalPlaces: 0)
```

By default, your app's locale will be used to format the number suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('stock')
->numeric(locale: 'nl')
```

Alternatively, you can set the default locale used across your app using the `Infolist::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Infolists\Infolist;

Infolist::$defaultNumberLocale = 'nl';
```

Currency formatting

The `money()` method allows you to easily format monetary values, in any currency:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('price')
->money('EUR')
```

There is also a `divideBy` argument for `money()` that allows you to divide the original value by a number before formatting it. This could be useful if your database stores the price in cents, for example:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('price')
->money('EUR', divideBy: 100)
```

By default, your app's locale will be used to format the money suitably. If you would like to customize the locale used, you can pass it to the `locale` argument:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('price')
->money('EUR', locale: 'nl')
```

Alternatively, you can set the default locale used across your app using the `Infolist::$defaultNumberLocale` method in the `boot()` method of a service provider:

```
use Filament\Infolists\Infolist;

Infolist::$defaultNumberLocale = 'nl';
```

Limiting text length

You may `limit()` the length of the entry's value:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
    ->limit(50)
```

You may also reuse the value that is being passed to `limit()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
    ->limit(50)
    ->tooltip(function (TextEntry $component): ?string {
        $state = $component->getState();

        if (strlen($state) <= $component->getCharacterLimit()) {
            return null;
        }

        // Only render the tooltip if the entry contents exceeds the length limit.
        return $state;
    })
```

Limiting word count

You may limit the number of `words()` displayed in the entry:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
    ->words(10)
```

Limiting text to a specific number of lines

You may want to limit text to a specific number of lines instead of limiting it to a fixed length. Clamping text to a number of lines is useful in responsive interfaces where you want to ensure a consistent experience across all screen sizes. This can be achieved using the `lineClamp()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
    ->lineClamp(2)
```

Listing multiple values

By default, if there are multiple values inside your text entry, they will be comma-separated. You may use the `listWithLineBreaks()` method to display them on new lines instead:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('authors.name')
->listWithLineBreaks()
```

Authors

Dan Harrin
Ryan Chandler
Zep Fietje
Dennis Koch
Adam Weston

Adding bullet points to the list

You may add a bullet point to each list item using the `bulleted()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('authors.name')
->listWithLineBreaks()
->bulleted()
```

Authors

- Dan Harrin
- Ryan Chandler
- Zep Fietje
- Dennis Koch
- Adam Weston

Limiting the number of values in the list

You can limit the number of values in the list using the `limitList()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('authors.name')
->listWithLineBreaks()
->limitList(3)
```

Expanding the limited list

You can allow the limited items to be expanded and collapsed, using the `expandableLimitedList()` method:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('authors.name')
->listWithLineBreaks()
->limitList(3)
->expandableLimitedList()
```

Please note that this is only a feature for `listWithLineBreaks()` or `bulleted()`, where each item is on its own line.

Using a list separator

If you want to "explode" a text string from your model into multiple list items, you can do so with the `separator()` method. This is useful for displaying comma-separated tags as badges, for example:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('tags')
->badge()
->separator(',',')
```

Rendering HTML

If your entry value is HTML, you may render it using `html()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
->html()
```

If you use this method, then the HTML will be sanitized to remove any potentially unsafe content before it is rendered. If you'd like to opt out of this behavior, you can wrap the HTML in an `HtmlString` object by formatting it:

```
use Filament\Infolists\Components\TextEntry;
use Illuminate\Support\HtmlString;

TextEntry::make('description')
->formatStateUsing(fn (string $state): HtmlString => new HtmlString($state))
```

Or, you can return a `view()` object from the `formatStateUsing()` method, which will also not be sanitized:

```
use Filament\Infolists\Components\TextEntry;
use Illuminate\Contracts\View\View;

TextEntry::make('description')
->formatStateUsing(fn (string $state): View => view(
    'filament.infolists.components.description-entry-content',
    ['state' => $state],
))
```

Rendering Markdown as HTML

If your entry value is Markdown, you may render it using `markdown()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('description')
->markdown()
```

Custom formatting

You may instead pass a custom formatting callback to `formatStateUsing()`, which accepts the `$state` of the entry, and optionally its `$record`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('status')
->formatStateUsing(fn (string $state): string => __("statuses.{${$state}}"))
```

Customizing the color

You may set a color for the text, either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('status')
->color('primary')
```

Status

Published

Adding an icon

Text entries may also have an icon:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('email')
->icon('heroicon-m-envelope')
```

Email

 dan@filamentphp.com

You may set the position of an icon using `iconPosition()`:

```
use Filament\Infolists\Components\TextEntry;
use Filament\Support\Enums\IconPosition;

TextEntry::make('email')
->icon('heroicon-m-envelope')
->iconPosition(IconPosition::After) // `IconPosition::Before` or `IconPosition::After`
```

Email

dan@filamentphp.com 

The icon color defaults to the text color, but you may customize the icon color separately using `iconColor()`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('email')
    ->icon('heroicon-m-envelope')
    ->iconColor('primary')
```

Email

 dan@filamentphp.com

Customizing the text size

Text columns have small font size by default, but you may change this to `TextEntrySize::ExtraSmall`, `TextEntrySize::Medium`, or `TextEntrySize::Large`.

For instance, you may make the text larger using `size(TextEntrySize::Large)`:

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('title')
    ->size(TextEntry\TextEntrySize::Large)
```

Title

What is Filament?

Customizing the font weight

Text entries have regular font weight by default, but you may change this to any of the following options:

`FontWeight::Thin`, `FontWeight::ExtraLight`, `FontWeight::Light`, `FontWeight::Medium`,
`FontWeight::SemiBold`, `FontWeight::Bold`, `FontWeight::ExtraBold` or `FontWeight::Black`.

For instance, you may make the font bold using `weight(FontWeight::Bold)`:

```
use Filament\Infolists\Components\TextEntry;
use Filament\Support\Enums\FontWeight;

TextEntry::make('title')
->weight(FontWeight::Bold)
```

Title

What is Filament?

Customizing the font family

You can change the text font family to any of the following options: `FontFamily::Sans`, `FontFamily::Serif` or
`FontFamily::Mono`.

For instance, you may make the font monospaced using `fontFamily(FontFamily::Mono)`:

```
use Filament\Support\Enums\FontFamily;
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->fontFamily(FontFamily::Mono)
```

API key

HGA3CH5AB345JD9MQ3

Allowing the text to be copied to the clipboard

You may make the text copyable, such that clicking on the entry copies the text to the clipboard, and optionally specify a custom confirmation message and duration in milliseconds. This feature only works when SSL is enabled for the app.

```
use Filament\Infolists\Components\TextEntry;

TextEntry::make('apiKey')
->label('API key')
->copyable()
->copyMessage('Copied!')
->copyMessageDuration(1500)
```

API key Copied!

HGA3CH5AB345JD9MQ3

Icon

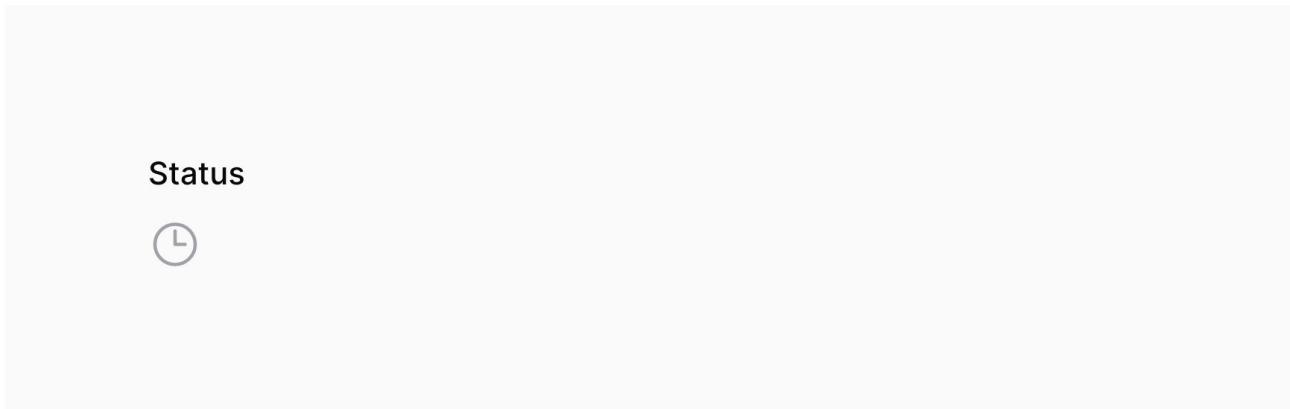
Overview

Icon entries render an `icon` representing their contents:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('status')
->icon(fn (string $state): string => match ($state) {
    'draft' => 'heroicon-o-pencil',
    'reviewing' => 'heroicon-o-clock',
    'published' => 'heroicon-o-check-circle',
})
```

In the function, `$state` is the value of the entry, and `$record` can be used to access the underlying Eloquent record.



Customizing the color

Icon entries may also have a set of icon colors, using the same syntax. They may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('status')
->color(fn (string $state): string => match ($state) {
    'draft' => 'info',
    'reviewing' => 'warning',
    'published' => 'success',
    default => 'gray',
})
```

In the function, `$state` is the value of the entry, and `$record` can be used to access the underlying Eloquent record.

Status



Customizing the size

The default icon size is `IconEntrySize::Large`, but you may customize the size to be either

`IconEntrySize::ExtraSmall`, `IconEntrySize::Small`, `IconEntrySize::Medium`,
`IconEntrySize::ExtraLarge` or `IconEntrySize::TwoExtraLarge`:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('status')
->size(IconEntry\IconEntrySize::Medium)
```

Status



Handling booleans

Icon entries can display a check or cross icon based on the contents of the database entry, either true or false, using the `boolean()` method:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('is_featured')
->boolean()
```

If this column in the model class is already cast as a `bool` or `boolean`, Filament is able to detect this, and you do not need to use `boolean()` manually.

Is featured



Is featured



Customizing the boolean icons

You may customize the icon representing each state. Icons are the name of a Blade component present. By default, [Heroicons](#) are installed:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('is_featured')
->boolean()
->trueIcon('heroicon-o-check-badge')
->falseIcon('heroicon-o-x-mark')
```

Is featured



Is featured



Customizing the boolean colors

You may customize the icon color representing each state. These may be either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
use Filament\Infolists\Components\IconEntry;

IconEntry::make('is_featured')
->boolean()
->trueColor('info')
->falseColor('warning')
```

Is featured



Is featured



Image

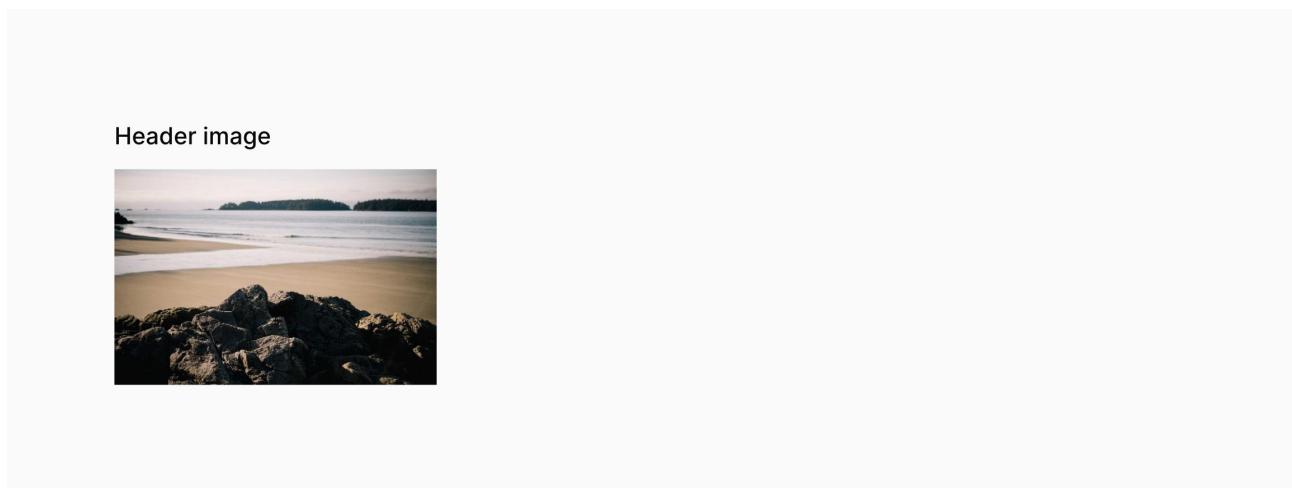
Overview

Images can be easily displayed within your infolist:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('header_image')
```

The entry must contain the path to the image, relative to the root directory of its storage disk, or an absolute URL to it.



Managing the image disk

By default, the `public` disk will be used to retrieve images. You may pass a custom disk name to the `disk()` method:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('header_image')
->disk('s3')
```

Private images

Filament can generate temporary URLs to render private images, you may set the `visibility()` to `private`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('header_image')
->visibility('private')
```

Customizing the size

You may customize the image size by passing a `width()` and `height()`, or both with `size()`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('header_image')
->width(200)

ImageEntry::make('header_image')
->height(50)

ImageEntry::make('author.avatar')
->size(40)
```

Square image

You may display the image using a 1:1 aspect ratio:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('author.avatar')
->height(40)
->square()
```

Author



Circular image

You may make the image fully rounded, which is useful for rendering avatars:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('author.avatar')
->height(40)
->circular()
```

Author

Adding a default image URL

You can display a placeholder image if one doesn't exist yet, by passing a URL to the `defaultImageUrl()` method:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('avatar')
->defaultImageUrl(url('/images/placeholder.png'))
```

Stacking images

You may display multiple images as a stack of overlapping images by using `stacked()`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
->height(40)
->circular()
->stacked()
```

Colleagues

Customizing the stacked ring width

The default ring width is `3`, but you may customize it to be from `0` to `8`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->ring(5)
```

Customizing the stacked overlap

The default overlap is 4, but you may customize it to be from 0 to 8:

```
use Filament\Infolists\Components\ImageEntry;

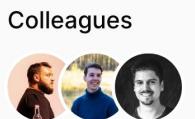
ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->overlap(2)
```

Setting a limit

You may limit the maximum number of images you want to display by passing `limit()`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->limit(3)
```



Showing the remaining images count

When you set a limit you may also display the count of remaining images by passing `limitedRemainingText()`.

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText()
```

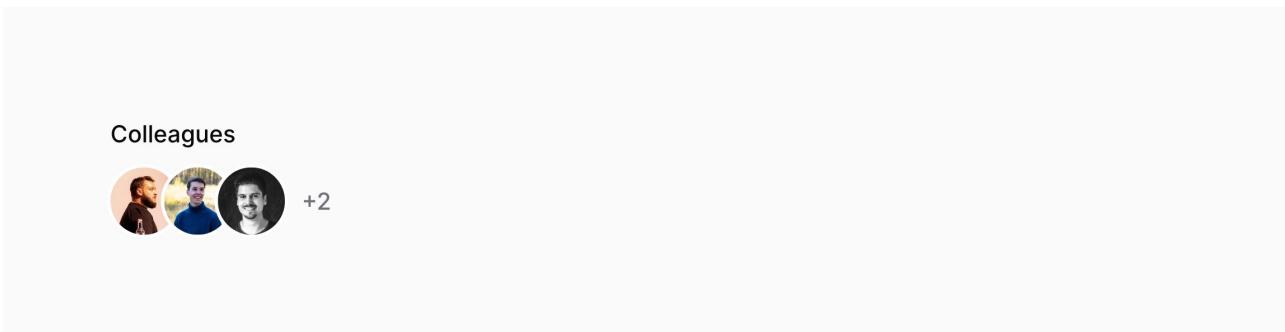


Showing the limited remaining text separately

By default, `limitedRemainingText()` will display the count of remaining images as a number stacked on the other images. If you prefer to show the count as a number after the images, you may use the `isSeparate: true` parameter:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText(isSeparate: true)
```



Customizing the limited remaining text size

By default, the size of the remaining text is `sm`. You can customize this to be `xs`, `md` or `lg` using the `size` parameter:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('colleagues.avatar')
    ->height(40)
    ->circular()
    ->stacked()
    ->limit(3)
    ->limitedRemainingText(size: 'lg')
```

Custom attributes

You may customize the extra HTML attributes of the image using `extraImgAttributes()`:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('logo')
    ->extraImgAttributes([
        'alt' => 'Logo',
        'loading' => 'lazy',
    ]),
```

Prevent file existence checks

When the infolist is loaded, it will automatically detect whether the images exist. This is all done on the backend. When using remote storage with many images, this can be time-consuming. You can use the `checkFileExistence(false)` method to disable this feature:

```
use Filament\Infolists\Components\ImageEntry;

ImageEntry::make('attachment')
    ->checkFileExistence(false)
```

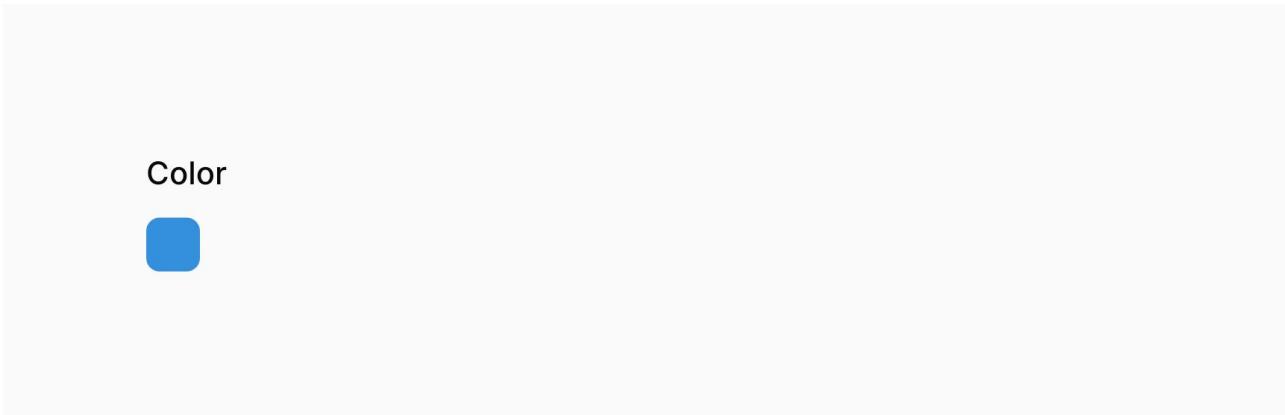
Color

Overview

The color entry allows you to show the color preview from a CSS color definition, typically entered using the color picker field, in one of the supported formats (HEX, HSL, RGB, RGBA).

```
use Filament\Infolists\Components\ColorEntry;

ColorEntry::make('color')
```



Allowing the color to be copied to the clipboard

You may make the color copyable, such that clicking on the preview copies the CSS value to the clipboard, and optionally specify a custom confirmation message and duration in milliseconds. This feature only works when SSL is enabled for the app.

```
use Filament\Infolists\Components\ColorEntry;

ColorEntry::make('color')
    ->copyable()
    ->copyMessage('Copied!')
    ->copyMessageDuration(1500)
```

Copied!



Key Value

Overview

The key-value entry allows you to render key-value pairs of data, from a one-dimensional JSON object / PHP array.

```
use Filament\Infolists\Components\KeyValueEntry;

KeyValueEntry::make('meta')
```

Meta

Key	Value
description	Filament is a collection of Laravel packages
og:type	website
og:site_name	Filament

If you're saving the data in Eloquent, you should be sure to add an `array` `cast` to the model property:

```
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $casts = [
        'meta' => 'array',
    ];

    // ...
}
```

Customizing the key column's label

You may customize the label for the key column using the `keyLabel()` method:

```
use Filament\Infolists\Components\KeyValueEntry;

KeyValueEntry::make('meta')
->keyLabel('Property name')
```

Customizing the value column's label

You may customize the label for the value column using the `valueLabel()` method:

```
use Filament\Infolists\Components\KeyValueEntry;

KeyValueEntry::make('meta')
->valueLabel('Property value')
```

Repeatable

Overview

The repeatable entry allows you to repeat a set of entries and layout components for items in an array or relationship.

```
use Filament\Infolists\Components\RepeatableEntry;
use Filament\Infolists\Components\TextEntry;

RepeatableEntry::make('comments')
    ->schema([
        TextEntry::make('author.name'),
        TextEntry::make('title'),
        TextEntry::make('content')
            ->columnSpan(2),
    ])
    ->columns(2)
```

As you can see, the repeatable entry has an embedded `schema()` which gets repeated for each item.

Comments

Author	Title
--------	-------

Jane Doe	Wow!
----------	------

Content

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Author	Title
--------	-------

John Doe	This isn't working. Help!
----------	---------------------------

Content

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

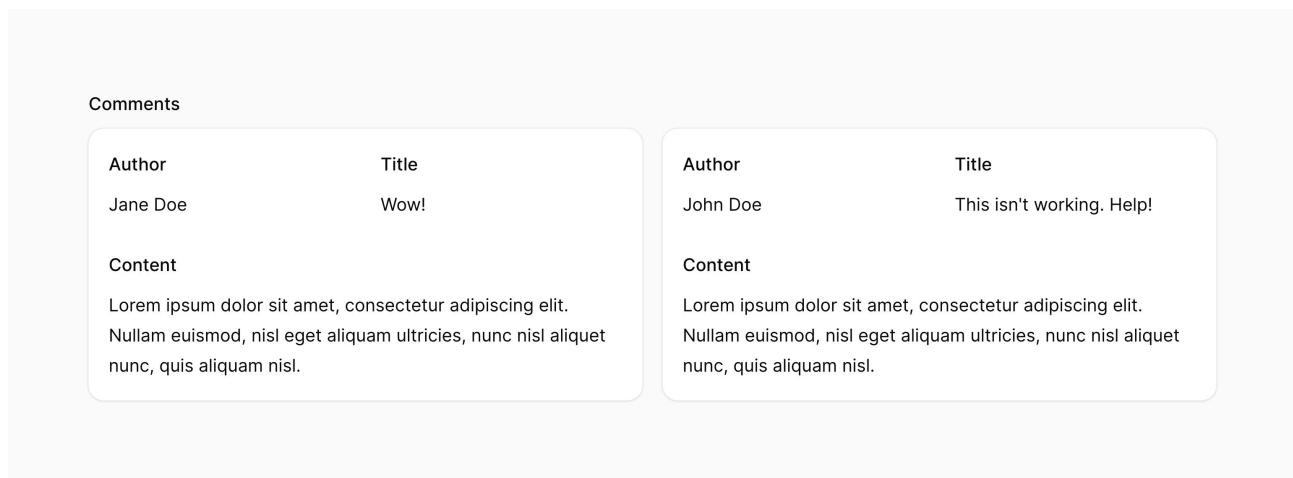
Grid layout

You may organize repeatable items into columns by using the `grid()` method:

```
use Filament\Infolists\Components\RepeatableEntry;

RepeatableEntry::make('comments')
    ->schema([
        // ...
    ])
    ->grid(2)
```

This method accepts the same options as the `columns()` method of the `grid`. This allows you to responsively customize the number of grid columns at various breakpoints.



Removing the styled container

By default, each item in a repeatable entry is wrapped in a container styled as a card. You may remove the styled container using `contained()`:

```
use Filament\Infolists\Components\RepeatableEntry;

RepeatableEntry::make('comments')
->schema([
    // ...
])
->contained(false)
```

Custom

View entries

You may render a custom view for an entry using the `view()` method:

```
use Filament\Infolists\Components\ViewEntry;

ViewEntry::make('status')
->view('filament.infolists.entries.status-switcher')
```

This assumes that you have a `resources/views/filament/infolists/entries/status-switcher.blade.php` file.

Custom classes

You may create your own custom entry classes and entry views, which you can reuse across your project, and even release as a plugin to the community.

If you're just creating a simple custom entry to use once, you could instead use a [view entry](#) to render any custom Blade file.

To create a custom entry class and view, you may use the following command:

```
php artisan make:infolist-entry StatusSwitcher
```

This will create the following entry class:

```
use Filament\Infolists\Components\Entry;

class StatusSwitcher extends Entry
{
    protected string $view = 'filament.infolists.entries.status-switcher';
}
```

It will also create a view file at `resources/views/filament/infolists/entries/status-switcher.blade.php`.

Accessing the state

Inside your view, you may retrieve the state of the entry using the `$getState()` function:

```
<div>
{{ $getState() }}
</div>
```

Accessing the Eloquent record

Inside your view, you may access the Eloquent record using the `$getRecord()` function:

```
<div>
{{ $getRecord() ->name }}</div>
```

Layout

Getting Started

Overview

Infolists are not limited to just displaying entries. You can also use "layout components" to organize them into an infinitely nestable structure.

Layout component classes can be found in the `Filament\Infolists\Components` namespace. They reside within the schema of your infolist, alongside any [entries](#).

Components may be created using the static `make()` method. Usually, you will then define the child component `schema()` to display inside:

```
use Filament\Infolists\Components\Grid;

Grid::make(2
    ->schema([
        // ...
    ]))
```

Available layout components

Filament ships with some layout components, suitable for arranging your form fields depending on your needs:

- [Grid](#)
- [Fieldset](#)
- [Tabs](#)
- [Section](#)
- [Split](#)

You may also [create your own custom layout components](#) to organize fields in whatever way you wish.

Setting an ID

You may define an ID for the component using the `id()` method:

```
use Filament\Infolists\Components\Section;

Section::make()
    ->id('main-section')
```

Adding extra HTML attributes

You can pass extra HTML attributes to the component, which will be merged onto the outer DOM element. Pass an array of attributes to the `extraAttributes()` method, where the key is the attribute name and the value is the attribute value:

```
use Filament\Infolists\Components\Group;

Section::make()
->extraAttributes(['class' => 'custom-section-style'])
```

Classes will be merged with the default classes, and any other attributes will override the default attributes.

Global settings

If you wish to change the default behavior of a component globally, then you can call the static `configureUsing()` method inside a service provider's `boot()` method, to which you pass a Closure to modify the component using. For example, if you wish to make all section components have [2 columns](#) by default, you can do it like so:

```
use Filament\Infolists\Components\Section;

Section::configureUsing(function (Section $section): void {
    $section
        ->columns(2);
});
```

Of course, you are still able to overwrite this on each field individually:

```
use Filament\Infolists\Components\Section;

Section::make()
->columns(1)
```

Grid

Overview

Filament's grid system allows you to create responsive, multi-column layouts using any layout component.

Responsively setting the number of grid columns

All layout components have a `columns()` method that you can use in a couple of different ways:

- You can pass an integer like `columns(2)`. This integer is the number of columns used on the `lg` breakpoint and higher. All smaller devices will have just 1 column.
- You can pass an array, where the key is the breakpoint and the value is the number of columns. For example, `columns(['md' => 2, 'xl' => 4])` will create a 2 column layout on medium devices, and a 4 column layout on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

Breakpoints (`sm`, `md`, `lg`, `xl`, `2xl`) are defined by Tailwind, and can be found in the [Tailwind documentation](#).

Controlling how many columns a component should span

In addition to specifying how many columns a layout component should have, you may also specify how many columns a component should fill within the parent grid, using the `columnSpan()` method. This method accepts an integer or an array of breakpoints and column spans:

- `columnSpan(2)` will make the component fill up to 2 columns on all breakpoints.
- `columnSpan(['md' => 2, 'xl' => 4])` will make the component fill up to 2 columns on medium devices, and up to 4 columns on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.
- `columnSpan('full')` or `columnSpanFull()` or `columnSpan(['default' => 'full'])` will make the component fill the full width of the parent grid, regardless of how many columns it has.

An example of a responsive grid layout

In this example, we have an infolist with a `section` layout component. Since all layout components support the `columns()` method, we can use it to create a responsive grid layout within the section itself.

We pass an array to `columns()` as we want to specify different numbers of columns for different breakpoints. On devices smaller than the `sm` [Tailwind breakpoint](#), we want to have 1 column, which is default. On devices larger than the `sm` breakpoint, we want to have 3 columns. On devices larger than the `xl` breakpoint, we want to have 6 columns. On devices larger than the `2xl` breakpoint, we want to have 8 columns.

Inside the section, we have a `text entry`. Since text entries are infolist components and all form components have a `columnSpan()` method, we can use it to specify how many columns the text entry should fill. On devices smaller than the `sm` breakpoint, we want the text entry to fill 1 column, which is default. On devices larger than the `sm` breakpoint, we want the text entry to fill 2 columns. On devices larger than the `xl` breakpoint, we want the text entry to fill 3 columns. On devices larger than the `2xl` breakpoint, we want the text entry to fill 4 columns.

```
use Filament\Infolists\Components\Section;
use Filament\Infolists\Components\TextEntry;

Section::make()
->columns([
    'sm' => 3,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    TextEntry::make('name')
        ->columnSpan([
            'sm' => 2,
            'xl' => 3,
            '2xl' => 4,
        ]),
    // ...
])
```

Grid component

All layout components support the `columns()` method, but you also have access to an additional `Grid` component. If you feel that your form schema would benefit from an explicit grid syntax with no extra styling, it may be useful to you. Instead of using the `columns()` method, you can pass your column configuration directly to `Grid::make()`:

```
use Filament\Infolists\Components\Grid;

Grid::make([
    'default' => 1,
    'sm' => 2,
    'md' => 3,
    'lg' => 4,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    // ...
])
```

Setting the starting column of a component in a grid

If you want to start a component in a grid at a specific column, you can use the `columnStart()` method. This method accepts an integer, or an array of breakpoints and which column the component should start at:

- `columnStart(2)` will make the component start at column 2 on all breakpoints.
- `columnStart(['md' => 2, 'xl' => 4])` will make the component start at column 2 on medium devices, and at column 4 on extra large devices. The default breakpoint for smaller devices uses 1 column, unless you use a `default` array key.

```
use Filament\Infolists\Components\Section;
use Filament\Infolists\Components\TextEntry;

Section::make()
->columns([
    'sm' => 3,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    TextEntry::make('name')
        ->columnStart([
            'sm' => 2,
            'xl' => 3,
            '2xl' => 4,
        ]),
    // ...
])
```

In this example, the grid has 3 columns on small devices, 6 columns on extra large devices, and 8 columns on extra extra large devices. The text entry will start at column 2 on small devices, column 3 on extra large devices, and column 4 on extra extra large devices. This is essentially producing a layout whereby the text entry always starts halfway through the grid, regardless of how many columns the grid has.

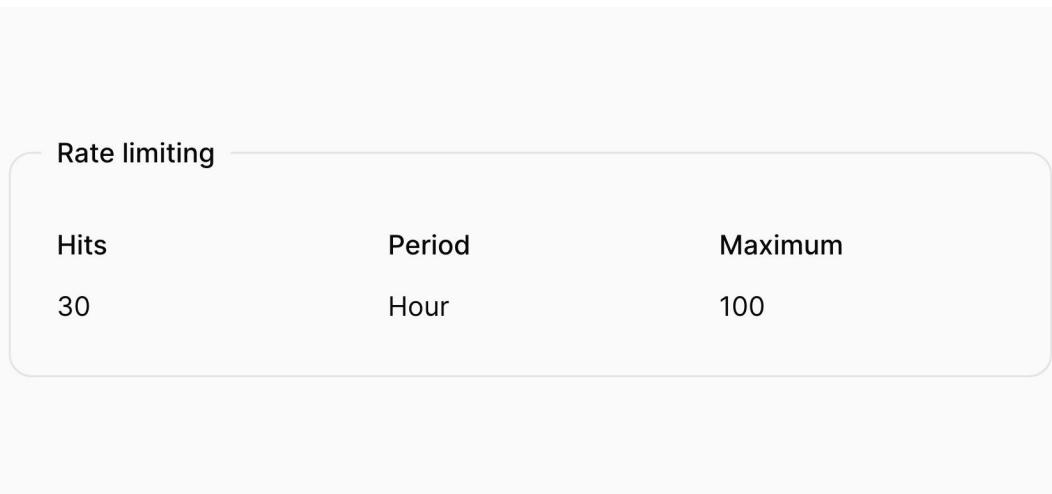
Fieldset

Overview

You may want to group entries into a Fieldset. Each fieldset has a label, a border, and a two-column grid by default:

```
use Filament\Infolists\Components\Fieldset;

Fieldset::make('Label')
->schema([
    // ...
])
```



Using grid columns within a fieldset

You may use the `columns()` method to customize the `grid` within the fieldset:

```
use Filament\Infolists\Components\Fieldset;

Fieldset::make('Label')
->schema([
    // ...
])
->columns(3)
```

Tabs

Overview

Some infolists can be long and complex. You may want to use tabs to reduce the number of components that are visible at once:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
    ->tabs([
        Tabs\Tab::make('Tab 1')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 2')
            ->schema([
                // ...
            ]),
        Tabs\Tab::make('Tab 3')
            ->schema([
                // ...
            ]),
    ])
)
```

Rate Limiting Proxy Meta

Hits	Period	Maximum
30	Hour	100

Notes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Setting the default active tab

The first tab will be open by default. You can change the default open tab using the `activeTab()` method:

```
use Filament\Infolists\Components\tabs;

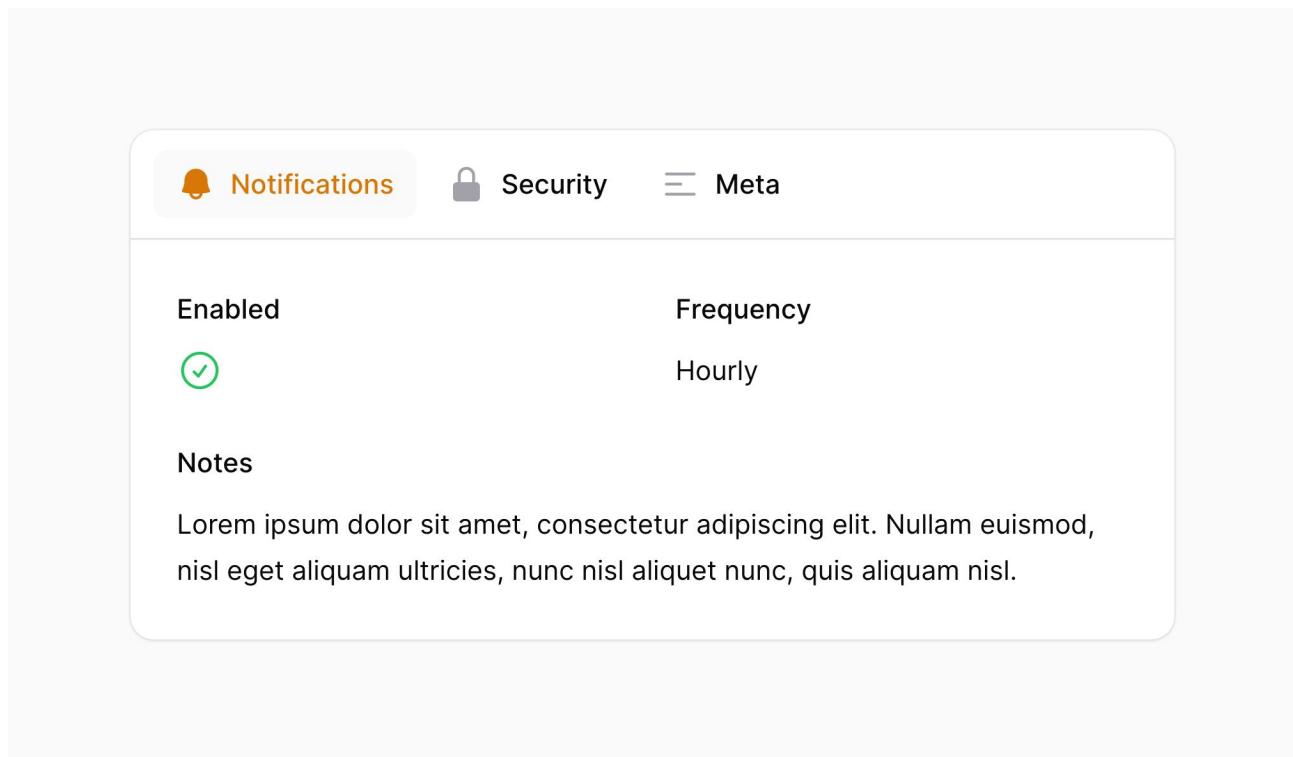
Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->activeTab(2)
```

Setting a tab icon

Tabs may have an `icon`, which you can set using the `icon()` method:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-m-bell')
        ->schema([
            // ...
        ]),
    // ...
])
```

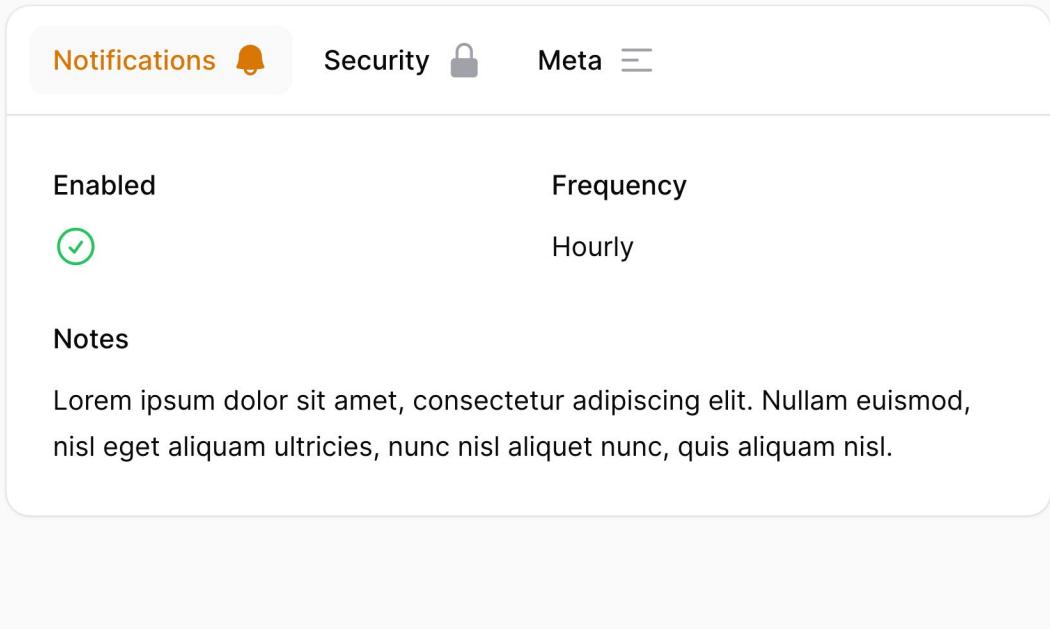


Setting the tab icon position

The icon of the tab may be positioned before or after the label using the `iconPosition()` method:

```
use Filament\Infolists\Components\tabs;
use Filament\Support\Enums\IconPosition;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-m-bell')
        ->iconPosition(IconPosition::After)
        ->schema([
            // ...
        ]),
        // ...
    ])
])
```

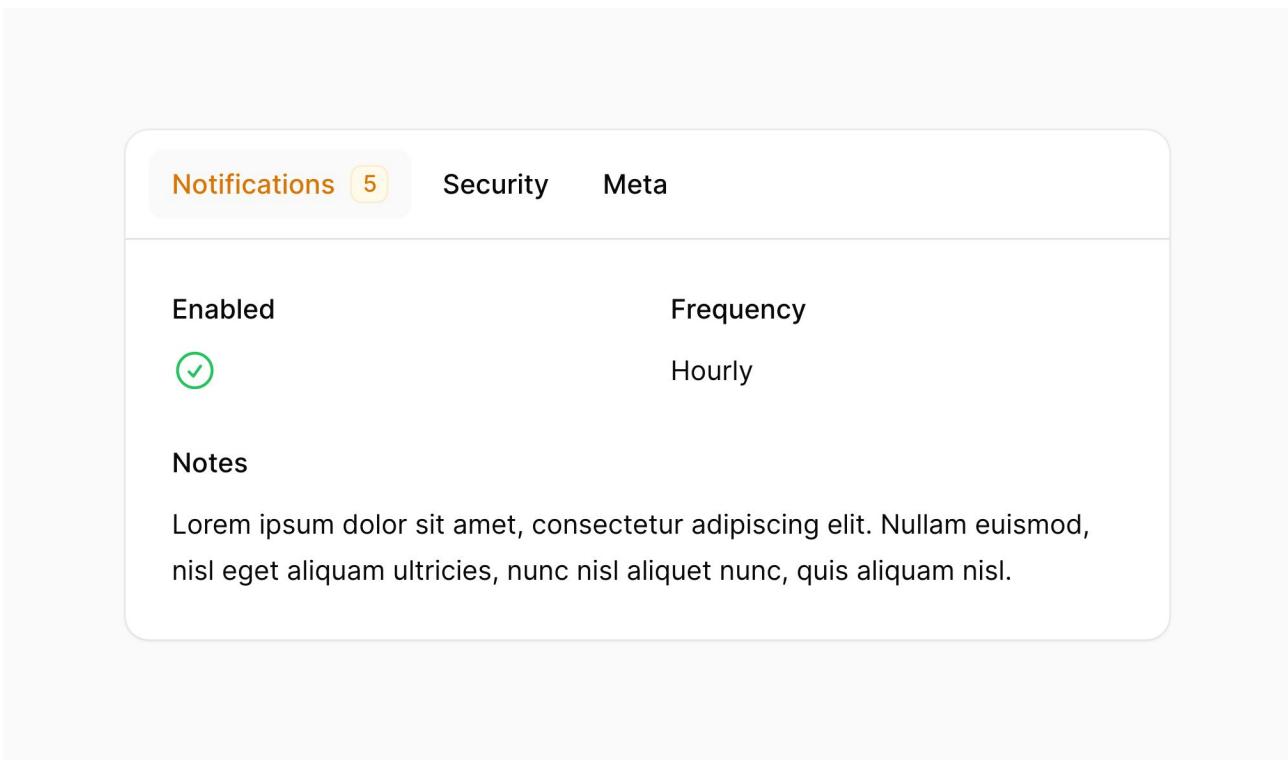


Setting a tab badge

Tabs may have a badge, which you can set using the `badge()` method:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->badge(5)
        ->schema([
            // ...
        ]),
        // ...
    ])
])
```



If you'd like to change the color for a badge, you can use the `badgeColor()` method:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Notifications')
        ->badge(5)
        ->badgeColor('success')
        ->schema([
            // ...
        ]),
        // ...
    ])
])
```

Using grid columns within a tab

You may use the `columns()` method to customize the `grid` within the tab:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ])
        ->columns(3),
        // ...
    ])
])
```

Removing the styled container

By default, tabs and their content are wrapped in a container styled as a card. You may remove the styled container using `contained()`:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->contained(false)
```

Persisting the current tab

By default, the current tab is not persisted in the browser's local storage. You can change this behavior using the `persistTab()` method. You must also pass in a unique `id()` for the tabs component, to distinguish it from all other sets of tabs in the app. This ID will be used as the key in the local storage to store the current tab:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    // ...
])
->persistent()
->id('order-tabs')
```

Persisting the current tab in the URL's query string

By default, the current tab is not persisted in the URL's query string. You can change this behavior using the `persistTabInQueryString()` method:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->persistentTabInQueryString()
```

By default, the current tab is persisted in the URL's query string using the `tab` key. You can change this key by passing it to the `persistentTabInQueryString()` method:

```
use Filament\Infolists\Components\tabs;

Tabs::make('Tabs')
->tabs([
    Tabs\Tab::make('Tab 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Tab 3')
        ->schema([
            // ...
        ]),
])
->persistentTabInQueryString('settings-tab')
```

Section

Overview

You may want to separate your entries into sections, each with a heading and description. To do this, you can use a section component:

```
use Filament\Infolists\Components\Section;

Section::make('Rate limiting')
    ->description('Prevent abuse by limiting the number of requests per period')
    ->schema([
        // ...
    ])
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour	100

Notes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

You can also use a section without a header, which just wraps the components in a simple card:

```
use Filament\Infolists\Components\Section;

Section::make()
    ->schema([
        // ...
    ])
```

Hits	Period	Maximum
30	Hour	100

Notes

 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod,
 nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Adding actions to the section's header or footer

Sections can have actions in their [header](#) or [footer](#).

Adding actions to the section's header

You may add [actions](#) to the section's header using the `headerActions()` method:

```
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\Section;

Section::make('Rate limiting')
    ->headerActions([
        Action::make('edit')
            ->action(function () {
                // ...
            }),
    ])
    ->schema([
        // ...
    ])
)
```

Rate limiting

[Edit](#)

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour	100

Notes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Make sure the section has a heading or ID

Adding actions to the section's footer

In addition to [header actions](#), you may add [actions](#) to the section's footer using the `footerActions()` method:

```
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\Section;

Section::make('Rate limiting')
    ->footerActions([
        Action::make('edit')
            ->action(function () {
                // ...
            }),
        ])
    ->schema([
        // ...
    ])
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour	100

Notes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

[Edit](#)

[Make sure the section has a heading or ID](#)

Aligning section footer actions

Footer actions are aligned to the inline start by default. You may customize the alignment using the `footerActionsAlignment()` method:

```
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\Section;
use Filament\Support\Enums\Alignment;

Section::make('Rate limiting')
->footerActions([
    Action::make('edit')
        ->action(function () {
            // ...
        }),
])
->footerActionsAlignment(Alignment::End)
->schema([
    // ...
])
```

Adding actions to a section without heading

If your section does not have a heading, Filament has no way of locating the action inside it. In this case, you must pass a unique `id()` to the section:

```
use Filament\Infolists\Components\Section;

Section::make()
->id('rateLimitingSection')
->headerActions([
    // ...
])
->schema([
    // ...
])
```

Adding an icon to the section's header

You may add an icon to the section's header using the `icon()` method:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
->description('The items you have selected for purchase')
->icon('heroicon-m-shopping-bag')
->schema([
    // ...
])
```

Product	Quantity
Filament t-shirt	3

Product	Quantity
Filament hoodie	1

Special order notes

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Positioning the heading and description aside

You may use the `aside()` method to align the heading and description on the left, and the infolist components inside a card on the right:

```
use Filament\Infolists\Components\Section;

Section::make('Rate limiting')
->description('Prevent abuse by limiting the number of requests per period')
->aside()
->schema([
    // ...
])
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits

30

Period

Hour

Maximum

100

Notes

 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Collapsing sections

Sections may be `collapsible()` to optionally hide content in long infolists:

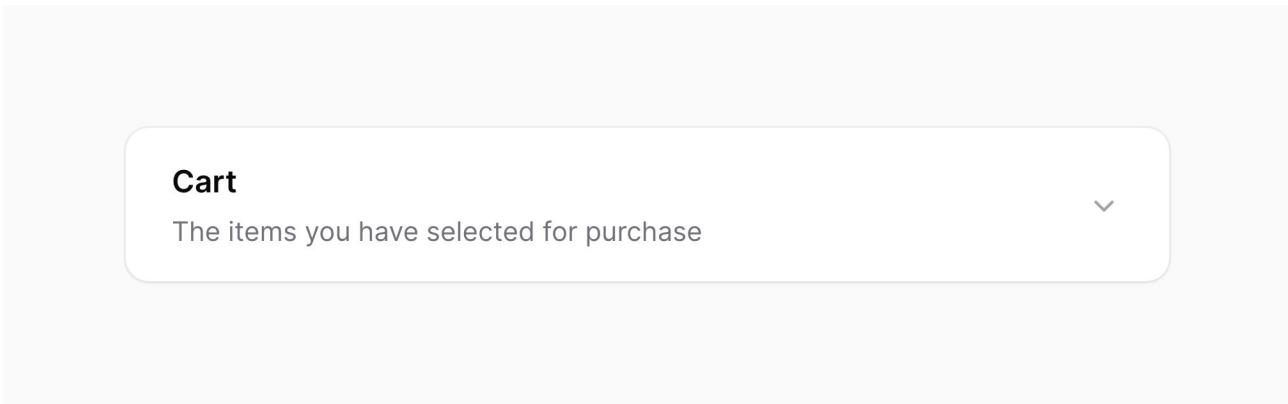
```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
```

Your sections may be `collapsed()` by default:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsed()
```



Persisting collapsed sections

You can persist whether a section is collapsed in local storage using the `persistCollapsed()` method, so it will remain collapsed when the user refreshes the page:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
    ->persistentCollapsed()
```

To persist the collapse state, the local storage needs a unique ID to store the state. This ID is generated based on the heading of the section. If your section does not have a heading, or if you have multiple sections with the same heading that you do not want to collapse together, you can manually specify the `id()` of that section to prevent an ID conflict:

```
use Filament\Infolists\Components\Section;

Section::make('Cart')
    ->description('The items you have selected for purchase')
    ->schema([
        // ...
    ])
    ->collapsible()
    ->persistentCollapsed()
    ->id('order-cart')
```

Compact section styling

When nesting sections, you can use a more compact styling:

```
use Filament\Infolists\Components\Section;

Section::make('Rate limiting')
    ->description('Prevent abuse by limiting the number of requests per period')
    ->schema([
        // ...
    ])
    ->compact()
```

Rate limiting

Prevent abuse by limiting the number of requests per period

Hits	Period	Maximum
30	Hour	100

Notes

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam euismod, nisl eget aliquam ultricies, nunc nisl aliquet nunc, quis aliquam nisl.

Using grid columns within a section

You may use the `columns()` method to easily create a grid within the section:

```
use Filament\Infolists\Components\Section;

Section::make('Heading')
    ->schema([
        // ...
    ])
    ->columns(2)
```

Split

Overview

The `Split` component allows you to define layouts with flexible widths, using flexbox.

```
use Filament\Infolists\Components\Section;
use Filament\Infolists\Components\Split;
use Filament\Infolists\Components\TextEntry;
use Filament\Support\Enums\FontWeight;

Split::make([
    Section::make([
        TextEntry::make('title')
            ->weight(FontWeight::Bold),
        TextEntry::make('content')
            ->markdown(),
            ->prose(),
    ]),
    Section::make([
        TextEntry::make('created_at')
            ->dateTime(),
        TextEntry::make('published_at')
            ->dateTime(),
    ]) ->grow(false),
]) ->from('md')
```

In this example, the first section will `grow()` to consume available horizontal space, without affecting the amount of space needed to render the second section. This creates a sidebar effect.

The `from()` method is used to control the [Tailwind breakpoint](#) (`sm`, `md`, `lg`, `xl`, `2xl`) at which the split layout should be used. In this example, the split layout will be used on medium devices and larger. On smaller devices, the sections will stack on top of each other.

Title**What is Filament?****Content**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non dui eu augue tempor finibus. Vivamus tincidunt malesuada volutpat. Donec ornare euismod est id cursus. Donec dolor nisl, dignissim vitae vulputate accumsan, consequat a lorem.

Praesent nec nulla ut erat mattis tincidunt. Donec dictum nibh at consequat finibus. Donec dignissim velit euismod dui lobortis, ut cursus ante auctor. Donec id placerat felis. Nulla dolor arcu, sodales nec sapien ut, laoreet rhoncus risus. Interdum et malesuada fames ac ante ipsum primis in faucibus.

Created at

Jan 1, 2022 14:10:32

Published at

Jul 2, 2023 23:29:24

Custom

View components

Aside from [building custom layout components](#), you may create "view" components which allow you to create custom layouts without extra PHP classes.

```
use Filament\Infolists\Components\View;

View::make('filament.infolists.components.box')
```

This assumes that you have a `resources/views/filament/infolists/components/box.blade.php` file.

Custom layout classes

You may create your own custom component classes and views, which you can reuse across your project, and even release as a plugin to the community.

If you're just creating a simple custom component to use once, you could instead use a [view component](#) to render any custom Blade file.

To create a custom column class and view, you may use the following command:

```
php artisan make:infolist-layout Box
```

This will create the following layout component class:

```
use Filament\Infolists\Components\Component;

class Box extends Component
{
    protected string $view = 'filament.infolists.components.box';

    public static function make(): static
    {
        return app(static::class);
    }
}
```

It will also create a view file at `resources/views/filament/infolists/components/box.blade.php`.

Rendering the component's schema

Inside your view, you may render the component's `schema()` using the `$getChildComponentContainer()` function:

```
<div>
    {{ $getChildComponentContainer() }}
</div>
```

Accessing the Eloquent record

Inside your view, you may access the Eloquent record using the `$getRecord()` function:

```
<div>
    {{ $getRecord() ->name  } }
</div>
```

Actions

Overview

Filament's infolists can use [Actions](#). They are buttons that can be added to any infolist component. Also, you can [render anonymous sets of actions](#) on their own, that are not attached to a particular infolist component.

Defining a infolist component action

Action objects inside an infolist component are instances of [Filament/Infolists/Components/Actions/Action](#). You must pass a unique name to the action's `make()` method, which is used to identify it amongst others internally within Filament. You can [customize the trigger button](#) of an action, and even [open a modal](#) with little effort:

```
use App\Actions\ResetStars;
use Filament\Infolists\Components\Actions\Action;

Action::make('resetStars')
    ->icon('heroicon-m-x-mark')
    ->color('danger')
    ->requiresConfirmation()
    ->action(function (ResetStars $resetStars) {
        $resetStars();
    })
}
```

Adding an affix action to a entry

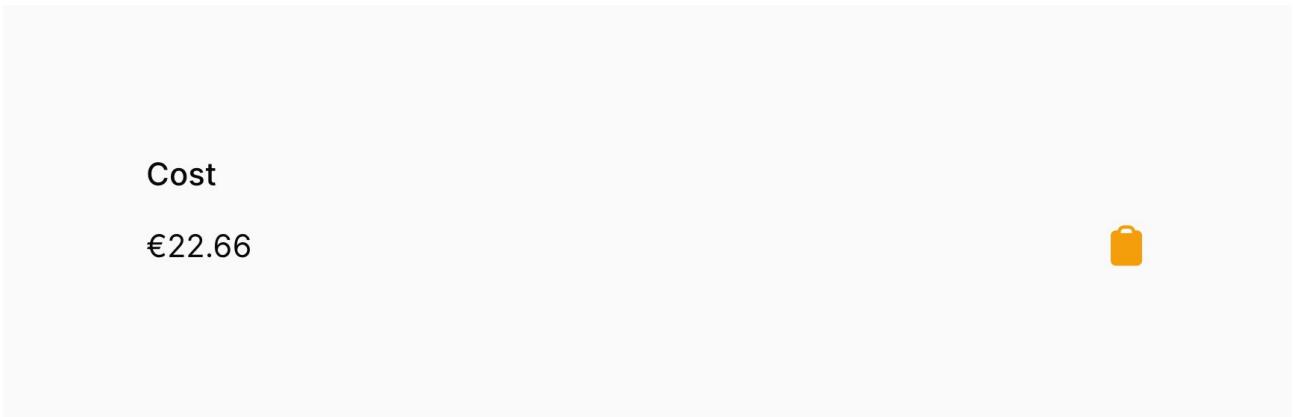
Certain entries support "affix actions", which are buttons that can be placed before or after its content. The following entries support affix actions:

- [Text entry](#)

To define an affix action, you can pass it to either `prefixAction()` or `suffixAction()`:

```
use App\Models\Product;
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\TextEntry;

TextEntry::make('cost')
    ->prefix('€')
    ->suffixAction(
        Action::make('copyCostToPrice')
            ->icon('heroicon-m-clipboard')
            ->requiresConfirmation()
            ->action(function (Product $record) {
                $record->price = $record->cost;
                $record->save();
            })
    )
)
```



Passing multiple affix actions to a entry

You may pass multiple affix actions to an entry by passing them in an array to either `prefixActions()` or `suffixActions()`. Either method can be used, or both at once, Filament will render all the registered actions in order:

```
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\TextEntry;

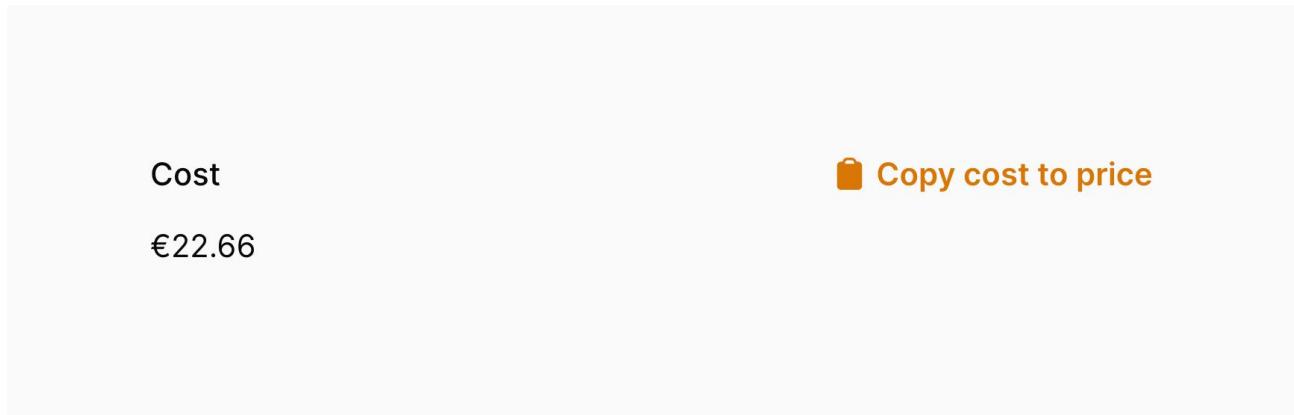
TextEntry::make('cost')
    ->prefix('€')
    ->prefixActions([
        Action::make('...'),
        Action::make('...'),
        Action::make('...'),
    ])
    ->suffixActions([
        Action::make('...'),
        Action::make('...'),
    ])
)
```

Adding a hint action to an entry

All entries support "hint actions", which are rendered aside the entry's `hint`. To add a hint action to a entry, you may pass it to `hintAction()`:

```
use App\Models\Product;
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\TextEntry;

TextEntry::make('cost')
    ->prefix('€')
    ->hintAction(
        Action::make('copyCostToPrice')
            ->icon('heroicon-m-clipboard')
            ->requiresConfirmation()
            ->action(function (Product $record) {
                $record->price = $record->cost;
                $record->save();
            })
    )
)
```



Passing multiple hint actions to a entry

You may pass multiple hint actions to a entry by passing them in an array to `hintActions()`. Filament will render all the registered actions in order:

```
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\TextEntry;

TextEntry::make('cost')
    ->prefix('€')
    ->hintActions([
        Action::make('...'),
        Action::make('...'),
        Action::make('...'),
    ])
)
```

Adding an action to a custom infolist component

If you wish to render an action within a custom infolist component, `ViewEntry` object, or `View` component object, you may do so using the `registerActions()` method:

```
use App\Models\Post;
use Filament\Forms\Components\TextInput;
use Filament\Infolists\Components\Actions\Action;
use Filament\Infolists\Components\ViewEntry;
use Filament\Infolists\Set;

ViewEntry::make('status')
    ->view('filament.infolists.entries.status-switcher')
    ->registerActions([
        Action::make('createStatus')
            ->form([
                TextInput::make('name')
                    ->required(),
            ])
            ->icon('heroicon-m-plus')
            ->action(function (array $data, Post $record) {
                $record->status()->create($data);
            }),
    ])
)
```

Now, to render the action in the view of the custom component, you need to call `[$getAction()]`, passing the name of the action you registered:

```
<div>
    <select></select>

    {{ $getAction('createStatus')  }}
</div>
```

Adding "anonymous" actions to an infolist without attaching them to a component

You may use an `Actions` component to render a set of actions anywhere in the infolist, avoiding the need to register them to any particular component:

```
use App\Actions\Star;
use App\Actions\ResetStars;
use Filament\Infolists\Components\Actions;
use Filament\Infolists\Components\Actions\Action;

Actions::make([
    Action::make('star')
        ->icon('heroicon-m-star')
        ->requiresConfirmation()
        ->action(function (Star $star) {
            $star();
        }),
    Action::make('resetStars')
        ->icon('heroicon-m-x-mark')
        ->color('danger')
        ->requiresConfirmation()
        ->action(function (ResetStars $resetStars) {
            $resetStars();
        }),
],
```



Making the independent infolist actions consume the full width of the infolist

You can stretch the independent infolist actions to consume the full width of the infolist using `fullWidth()`:

```
use Filament\Infolists\Components\Actions;

Actions::make([
    // ...
])->fullWidth(),
```

★ Star

✗ Reset stars

Controlling the horizontal alignment of independent infolist actions

Independent infolist actions are aligned to the start of the component by default. You may change this by passing

`Alignment::Center` or `Alignment::End` to `alignment()`:

```
use Filament\Infolists\Components\Actions;
use Filament\Support\Enums\Alignment;

Actions::make([
    // ...
])->alignment(Alignment::Center),
```

★ Star

✗ Reset stars

Controlling the vertical alignment of independent infolist actions

Independent infolist actions are vertically aligned to the start of the component by default. You may change this by

passing `Alignment::Center` or `Alignment::End` to `verticalAlignment()`:

```
use Filament\Infolists\Components\Actions;
use Filament\Support\Enums\VerticalAlignment;

Actions::make([
    // ...
])->verticalAlignment(VerticalAlignment::End),
```

Stars

4,572,100,479

★ Star

✗ Reset stars

Advanced

Inserting Livewire components into an infolist

You may insert a Livewire component directly into an infolist:

```
use Filament\Infolists\Components\Livewire;
use App\Livewire\Foo;

Livewire::make(Foo::class)
```

If you are rendering multiple of the same Livewire component, please make sure to pass a unique `key()` to each:

```
use Filament\Infolists\Components\Livewire;
use App\Livewire\Foo;

Livewire::make(Foo::class)
    ->key('foo-first')

Livewire::make(Foo::class)
    ->key('foo-second')

Livewire::make(Foo::class)
    ->key('foo-third')
```

Passing parameters to a Livewire component

You can pass an array of parameters to a Livewire component:

```
use Filament\Infolists\Components\Livewire;
use App\Livewire\Foo;

Livewire::make(Foo::class, ['bar' => 'baz'])
```

Now, those parameters will be passed to the Livewire component's `mount()` method:

```
class Foo extends Component
{
    public function mount(string $bar): void
    {
        // ...
    }
}
```

Alternatively, they will be available as public properties on the Livewire component:

```
class Foo extends Component
{
    public string $bar;
}
```

Accessing the current record in the Livewire component

You can access the current record in the Livewire component using the `$record` parameter in the `mount()` method, or the `$record` property:

```
use Illuminate\Database\Eloquent\Model;

class Foo extends Component
{
    public function mount(Model $record): void
    {
        // ...
    }

    // or

    public Model $record;
}
```

Lazy loading a Livewire component

You may allow the component to lazily load using the `lazy()` method:

```
use Filament\Infolists\Components\Livewire;
use App\LiveWire\Foo;

Livewire::make(Foo::class)->lazy()
```

Adding An Infolist To A Livewire Component

Setting up the Livewire component

First, generate a new Livewire component:

```
php artisan make:livewire ViewProduct
```

Then, render your Livewire component on the page:

```
@livewire('view-product')
```

Alternatively, you can use a full-page Livewire component:

```
use App\Livewire\ViewProduct;
use Illuminate\Support\Facades\Route;

Route::get('products/{product}', ViewProduct::class);
```

You must use the `InteractsWithInfolists` and `InteractsWithForms` traits, and implement the `HasInfolists` and `HasForms` interfaces on your Livewire component class:

```
use Filament\Forms\Concerns\InteractsWithForms;
use Filament\Forms\Contracts\HasForms;
use Filament\Infolists\Concerns\InteractsWithInfolists;
use Filament\Infolists\Contracts\HasInfolists;
use Livewire\Component;

class ViewProduct extends Component implements HasForms, HasInfolists
{
    use InteractsWithInfolists;
    use InteractsWithForms;

    // ...
}
```

Adding the infolist

Next, add a method to the Livewire component which accepts an `$infolist` object, modifies it, and returns it:

```
use Filament\Infolists\Infolist;

public function productInfolist(Infolist $infolist): Infolist
{
    return $infolist
        ->record($this->product)
        ->schema([
            // ...
        ]);
}
```

Finally, render the infolist in the Livewire component's view:

```
{{ $this->productInfolist }}
```

Passing data to the infolist

You can pass data to the infolist in two ways:

Either pass an Eloquent model instance to the `record()` method of the infolist, to automatically map all the model attributes and relationships to the entries in the infolist's schema:

```
use Filament\Infolists\Components\TextEntry;
use Filament\Infolists\Infolist;

public function productInfolist(Infolist $infolist): Infolist
{
    return $infolist
        ->record($this->product)
        ->schema([
            TextEntry::make('name'),
            TextEntry::make('category.name'),
            // ...
        ]);
}
```

Alternatively, you can pass an array of data to the `state()` method of the infolist, to manually map the data to the entries in the infolist's schema:

```
use Filament\Infolists\Components\TextEntry;
use Filament\Infolists\Infolist;

public function productInfolist(Infolist $infolist): Infolist
{
    return $infolist
        ->state([
            'name' => 'MacBook Pro',
            'category' => [
                'name' => 'Laptops',
            ],
            // ...
        ])
        ->schema([
            TextEntry::make('name'),
            TextEntry::make('category.name'),
            // ...
        ]);
}
```

Testing

Overview

All examples in this guide will be written using [Pest](#). To use Pest's Livewire plugin for testing, you can follow the installation instructions in the Pest documentation on plugins: [Livewire plugin for Pest](#). However, you can easily adapt this to PHPUnit.

Since the Infolist Builder works on Livewire components, you can use the [Livewire testing helpers](#). However, we have custom testing helpers that you can use with infolists:

Actions

You can call an action by passing its infolist component key, and then the name of the action to `callInfolistAction()`:

```
use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callInfolistAction('customer', 'send', infolistName: 'infolist');

    expect($invoice->refresh())
        ->isSent()->toBeTrue();
});

});
```

To pass an array of data into an action, use the `data` parameter:

```
use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callInfolistAction('customer', 'send', data: [
        'email' => $email = fake()->email(),
    ])
    ->assertHasNoInfolistActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($email);
});

});
```

If you ever need to only set an action's data without immediately calling it, you can use `setInfolistActionData()`:

```
use function Pest\Livewire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountInfolistAction('customer', 'send')
    ->setInfolistActionData('customer', 'send', data: [
        'email' => $email = fake()->email(),
    ])
});
});
```

Execution

To check if an action has been halted, you can use `assertInfolistActionHalted()`:

```
use function Pest\Livewire\livewire;

it('stops sending if invoice has no email address', function () {
    $invoice = Invoice::factory(['email' => null])->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callInfolistAction('customer', 'send')
    ->assertInfolistActionHalted('customer', 'send');
});
});
```

Errors

`assertHasNoInfolistActionErrors()` is used to assert that no validation errors occurred when submitting the action form.

To check if a validation error has occurred with the data, use `assertHasInfolistActionErrors()`, similar to `assertHasErrors()` in Livewire:

```
use function Pest\Livewire\livewire;

it('can validate invoice recipient email', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callInfolistAction('customer', 'send', data: [
        'email' => Str::random(),
    ])
    ->assertHasInfolistActionErrors(['email' => ['email']]);
});
});
```

To check if an action is pre-filled with data, you can use the `assertInfolistActionDataSet()` method:

```
use function Pest\Livewire\livewire;

it('can send invoices to the primary contact by default', function () {
    $invoice = Invoice::factory()->create();
    $recipientEmail = $invoice->company->primaryContact->email;

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountInfolistAction('customer', 'send')
    ->assertInfolistActionDataSet([
        'email' => $recipientEmail,
    ])
    ->callMountedInfolistAction()
    ->assertHasNoInfolistActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($recipientEmail);
});

});
```

Action state

To ensure that an action exists or doesn't in an infolist, you can use the `assertInfolistActionExists()` or `assertInfolistActionDoesNotExist()` method:

```
use function Pest\Livewire\livewire;

it('can send but not unsend invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertInfolistActionExists('customer', 'send')
    ->assertInfolistActionDoesNotExist('customer', 'unsend');
});
```

To ensure an action is hidden or visible for a user, you can use the `assertInfolistActionHidden()` or `assertInfolistActionVisible()` methods:

```
use function Pest\Livewire\livewire;

it('can only print customers', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertInfolistActionHidden('customer', 'send')
    ->assertInfolistActionVisible('customer', 'print');
});
```

To ensure an action is enabled or disabled for a user, you can use the `assertInfolistActionEnabled()` or `assertInfolistActionDisabled()` methods:

```
use function Pest\LiveWire\livewire;

it('can only print a customer for a sent invoice', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertInfolistActionDisabled('customer', 'send')
        ->assertInfolistActionEnabled('customer', 'print');
});
```

To check if an action is hidden to a user, you can use the `assertInfolistActionHidden()` method:

```
use function Pest\LiveWire\livewire;

it('can not send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertInfolistActionHidden('customer', 'send');
});
```

Button appearance

To ensure an action has the correct label, you can use `assertInfolistActionHasLabel()` and `assertInfolistActionDoesNotHaveLabel()`:

```
use function Pest\LiveWire\livewire;

it('send action has correct label', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertInfolistActionHasLabel('customer', 'send', 'Email Invoice')
        ->assertInfolistActionDoesNotHaveLabel('customer', 'send', 'Send');
});
```

To ensure an action's button is showing the correct icon, you can use `assertInfolistActionHasIcon()` or `assertInfolistActionDoesNotHaveIcon()`:

```
use function Pest\Livewire\livewire;

it('when enabled the send button has correct icon', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertInfolistActionEnabled('customer', 'send')
    ->assertInfolistActionHasIcon('customer', 'send', 'envelope-open')
    ->assertInfolistActionDoesNotHaveIcon('customer', 'send', 'envelope');
});
```

To ensure that an action's button is displaying the right color, you can use `assertInfolistActionHasColor()` or `assertInfolistActionDoesNotHaveColor()`:

```
use function Pest\Livewire\livewire;

it('actions display proper colors', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertInfolistActionHasColor('customer', 'delete', 'danger')
    ->assertInfolistActionDoesNotHaveColor('customer', 'print', 'danger');
});
```

URL

To ensure an action has the correct URL, you can use `assertInfolistActionHasUrl()`, `assertInfolistActionDoesNotHaveUrl()`, `assertInfolistActionShouldOpenUrlInNewTab()`, and `assertInfolistActionShouldNotOpenUrlInNewTab()`:

```
use function Pest\Livewire\livewire;

it('links to the correct Filament sites', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertInfolistActionHasUrl('customer', 'filament', 'https://filamentphp.com/')
    ->assertInfolistActionDoesNotHaveUrl('customer', 'filament',
    'https://github.com/filamentphp/filament')
    ->assertInfolistActionShouldOpenUrlInNewTab('customer', 'filament')
    ->assertInfolistActionShouldNotOpenUrlInNewTab('customer', 'github');
});
```

Chapter 7

Widgets

Installation

The Widgets package is pre-installed with the [Panel Builder](#). This guide is for using the Widgets package in a custom TALL Stack application (Tailwind, Alpine, Livewire, Laravel).

Requirements

Filament requires the following to run:

- PHP 8.1+
- Laravel v10.0+
- Livewire v3.0+

Installation

Require the Widgets package using Composer:

```
composer require filament/widgets:"^3.2" -W
```

New Laravel projects

To quickly get started with Filament in a new Laravel project, run the following commands to install [Livewire](#), [Alpine.js](#), and [Tailwind CSS](#):

Since these commands will overwrite existing files in your application, only run this in a new Laravel project!

```
php artisan filament:install --scaffold --widgets  
npm install  
npm run dev
```

Existing Laravel projects

Run the following command to install the Widgets package assets:

```
php artisan filament:install --widgets
```

Installing Tailwind CSS

Run the following command to install Tailwind CSS with the Tailwind Forms and Typography plugins:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography postcss postcss-nesting  
autoprefixer --save-dev
```

Create a new `tailwind.config.js` file and add the Filament `preset` (*includes the Filament color scheme and the required Tailwind plugins*):

```
import preset from './vendor/filament/support/tailwind.config.preset'

export default {
    presets: [preset],
    content: [
        './app/Filament/**/*.php',
        './resources/views/filament/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
    ],
}
```

Configuring styles

Add Tailwind's CSS layers to your `resources/css/app.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Create a `postcss.config.js` file in the root of your project and register Tailwind CSS, PostCSS Nesting and Autoprefixer as plugins:

```
export default {
    plugins: {
        'tailwindcss/nesting': 'postcss-nesting',
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

Automatically refreshing the browser

You may also want to update your `vite.config.js` file to refresh the page automatically when Livewire components are updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: [
                ...refreshPaths,
                'app/Livewire/**',
            ],
        }),
    ],
})
```

Compiling assets

Compile your new CSS and Javascript assets using `npm run dev`.

Configuring your layout

Create a new `resources/views/components/layouts/app.blade.php` layout file for Livewire components:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">

        <meta name="application-name" content="{{ config('app.name') }}">
        <meta name="csrf-token" content="{{ csrf_token() }}">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>{{ config('app.name') }}</title>

        <style>
            [x-cloak] {
                display: none !important;
            }
        </style>

        @filamentStyles
        @vite('resources/css/app.css')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @filamentScripts
        @vite('resources/js/app.js')
    </body>
</html>
```

Publishing configuration

You can publish the package configuration using the following command (optional):

```
php artisan vendor:publish --tag=filament-config
```

Upgrading

Filament automatically upgrades to the latest non-breaking version when you run `composer update`. After any updates, all Laravel caches need to be cleared, and frontend assets need to be republished. You can do this all at once using the `filament:upgrade` command, which should have been added to your `composer.json` file when you ran `filament:install` the first time:

```
"post-autoload-dump": [
    // ...
    "@php artisan filament:upgrade"
],
```

Please note that `filament:upgrade` does not actually handle the update process, as Composer does that already. If you're upgrading manually without a `post-autoload-dump` hook, you can run the command yourself:

```
composer update
```

```
php artisan filament:upgrade
```

Stats Overview

Overview

Filament comes with a "stats overview" widget template, which you can use to display a number of different stats in a single widget, without needing to write a custom view.

Start by creating a widget with the command:

```
php artisan make:filament-widget StatsOverview --stats-overview
```

This command will create a new `StatsOverview.php` file. Open it, and return `Stat` instances from the `getStats()` method:

```
<?php

namespace App\Filament\widgets;

use Filament\Widgets\StatsOverviewWidget as BaseWidget;
use Filament\Widgets\StatsOverviewWidget\Stat;

class StatsOverview extends BaseWidget
{
    protected function getStats(): array
    {
        return [
            Stat::make('Unique views', '192.1k'),
            Stat::make('Bounce rate', '21%'),
            Stat::make('Average time on page', '3:12'),
        ];
    }
}
```

Now, check out your widget in the dashboard.

Adding a description and icon to a stat

You may add a `description()` to provide additional information, along with a `descriptionIcon()`:

```
use Filament\Widgets\StatsOverviewWidget\Stat;

protected function getStats(): array
{
    return [
        Stat::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-m-arrow-trending-up'),
        Stat::make('Bounce rate', '21%')
            ->description('7% decrease')
            ->descriptionIcon('heroicon-m-arrow-trending-down'),
        Stat::make('Average time on page', '3:12')
            ->description('3% increase')
            ->descriptionIcon('heroicon-m-arrow-trending-up'),
    ];
}
```

The `descriptionIcon()` method also accepts a second parameter to put the icon before the description instead of after it:

```
use Filament\Support\Enums\IconPosition;
use Filament\Widgets\StatsOverviewWidget\Stat;

Stat::make('Unique views', '192.1k')
    ->description('32k increase')
    ->descriptionIcon('heroicon-m-arrow-trending-up', IconPosition::Before)
```

Changing the color of the stat

You may also give stats a `color()` (`danger`, `gray`, `info`, `primary`, `success` or `warning`):

```
use Filament\Widgets\StatsOverviewWidget\Stat;

protected function getStats(): array
{
    return [
        Stat::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-m-arrow-trending-up')
            ->color('success'),
        Stat::make('Bounce rate', '21%')
            ->description('7% increase')
            ->descriptionIcon('heroicon-m-arrow-trending-down')
            ->color('danger'),
        Stat::make('Average time on page', '3:12')
            ->description('3% increase')
            ->descriptionIcon('heroicon-m-arrow-trending-up')
            ->color('success'),
    ];
}
```

Adding extra HTML attributes to a stat

You may also pass extra HTML attributes to stats using `extraAttributes()`:

```
use Filament\Widgets\StatsOverviewWidget\Stat;

protected function getStats(): array
{
    return [
        Stat::make('Processed', '192.1k')
            ->color('success')
            ->extraAttributes([
                'class' => 'cursor-pointer',
                'wire:click' => "\$dispatch('setStatusFilter', { filter: 'processed' })",
            ]),
        // ...
    ];
}
```

In this example, we are deliberately escaping the `$` in `\$dispatch()` since this needs to be passed directly to the HTML, it is not a PHP variable.

Adding a chart to a stat

You may also add or chain a `chart()` to each stat to provide historical data. The `chart()` method accepts an array of data points to plot:

```
use Filament\Widgets\StatsOverviewWidget\Stat;

protected function getStats(): array
{
    return [
        Stat::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-m-arrow-trending-up')
            ->chart([7, 2, 10, 3, 15, 4, 17])
            ->color('success'),
        // ...
    ];
}
```

Live updating stats (polling)

By default, stats overview widgets refresh their data every 5 seconds.

To customize this, you may override the `$pollingInterval` property on the class to a new interval:

```
protected static ?string $pollingInterval = '10s';
```

Alternatively, you may disable polling altogether:

```
protected static ?string $pollingInterval = null;
```

Disabling lazy loading

By default, widgets are lazy-loaded. This means that they will only be loaded when they are visible on the page.

To disable this behavior, you may override the `$isLazy` property on the widget class:

```
protected static bool $isLazy = false;
```

Charts

Overview

Filament comes with many "chart" widget templates, which you can use to display real-time, interactive charts.

Start by creating a widget with the command:

```
php artisan make:filament-widget BlogPostsChart --chart
```

There is a single `ChartWidget` class that is used for all charts. The type of chart is set by the `getType()` method. In this example, that method returns the string `'line'`.

The `protected static ?string $heading` variable is used to set the heading that describes the chart. If you need to set the heading dynamically, you can override the `getHeading()` method.

The `getData()` method is used to return an array of datasets and labels. Each dataset is a labeled array of points to plot on the chart, and each label is a string. This structure is identical to the [Chart.js](#) library, which Filament uses to render charts. You may use the [Chart.js documentation](#) to fully understand the possibilities to return from `getData()`, based on the chart type.

```
<?php

namespace App\Filament\widgets;

use Filament\widgets\ChartWidget;

class BlogPostsChart extends ChartWidget
{
    protected static ?string $heading = 'Blog Posts';

    protected function getData(): array
    {
        return [
            'datasets' => [
                [
                    'label' => 'Blog posts created',
                    'data' => [0, 10, 5, 2, 21, 32, 45, 74, 65, 45, 77, 89],
                ],
                [
                    'label' => 'Blog posts read',
                    'data' => [0, 10, 5, 2, 21, 32, 45, 74, 65, 45, 77, 89],
                ],
            ],
            'labels' => ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'],
        ];
    }

    protected function getType(): string
    {
        return 'line';
    }
}
```

Now, check out your widget in the dashboard.

Available chart types

Below is a list of available chart widget classes which you may extend, and their corresponding [Chart.js documentation](#) page, for inspiration on what to return from `getData()`:

- Bar chart - [Chart.js documentation](#)
- Bubble chart - [Chart.js documentation](#)
- Doughnut chart - [Chart.js documentation](#)
- Line chart - [Chart.js documentation](#)
- Pie chart - [Chart.js documentation](#)
- Polar area chart - [Chart.js documentation](#)
- Radar chart - [Chart.js documentation](#)
- Scatter chart - [Chart.js documentation](#)

Customizing the chart color

You can customize the color of the chart data by setting the `$color` property to either `danger`, `gray`, `info`, `primary`, `success` or `warning`:

```
protected static string $color = 'info';
```

If you're looking to customize the color further, or use multiple colors across multiple datasets, you can still make use of Chart.js's [color options](#) in the data:

```
protected function getData(): array
{
    return [
        'datasets' => [
            [
                'label' => 'Blog posts created',
                'data' => [0, 10, 5, 2, 21, 32, 45, 74, 65, 45, 77, 89],
                'backgroundColor' => '#36A2EB',
                'borderColor' => '#9BD0F5',
            ],
            [
                'label' => 'Blog posts updated',
                'data' => [1, 12, 7, 3, 18, 29, 36, 54, 41, 52, 63, 71],
                'backgroundColor' => '#E91E63',
                'borderColor' => '#C8D4FA',
            ],
            [
                'label' => 'Blog posts deleted',
                'data' => [0, 5, 2, 1, 10, 15, 20, 25, 30, 35, 40, 45],
                'backgroundColor' => '#F08080',
                'borderColor' => '#F5B7B1',
            ],
        ],
        'labels' => ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'],
    ];
}
```

Generating chart data from an Eloquent model

To generate chart data from an Eloquent model, Filament recommends that you install the `flowframe/laravel-trend` package. You can view the [documentation](#).

Here is an example of generating chart data from a model using the `laravel-trend` package:

```

use Flowframe\Trend\Trend;
use Flowframe\Trend\TrendValue;

protected function getData(): array
{
    $data = Trend::model(BlogPost::class)
        ->between(
            start: now()->startOfYear(),
            end: now()->endOfYear(),
        )
        ->perMonth()
        ->count();

    return [
        'datasets' => [
            [
                'label' => 'Blog posts',
                'data' => $data->map(fn (TrendValue $value) => $value->aggregate),
            ],
            'labels' => $data->map(fn (TrendValue $value) => $value->date),
        ];
}

```

Filtering chart data

You can set up chart filters to change the data shown on chart. Commonly, this is used to change the time period that chart data is rendered for.

To set a default filter value, set the `$filter` property:

```
public ?string $filter = 'today';
```

Then, define the `getFilters()` method to return an array of values and labels for your filter:

```

protected function getFilters(): ?array
{
    return [
        'today' => 'Today',
        'week' => 'Last week',
        'month' => 'Last month',
        'year' => 'This year',
    ];
}

```

You can use the active filter value within your `getData()` method:

```

protected function getData(): array
{
    $activeFilter = $this->filter;

    // ...
}

```

Live updating chart data (polling)

By default, chart widgets refresh their data every 5 seconds.

To customize this, you may override the `$pollingInterval` property on the class to a new interval:

```
protected static ?string $pollingInterval = '10s';
```

Alternatively, you may disable polling altogether:

```
protected static ?string $pollingInterval = null;
```

Setting a maximum chart height

You may place a maximum height on the chart to ensure that it doesn't get too big, using the `$maxHeight` property:

```
protected static ?string $maxHeight = '300px';
```

Setting chart configuration options

You may specify an `$options` variable on the chart class to control the many configuration options that the Chart.js library provides. For instance, you could turn off the `legend` for a line chart:

```
protected static ?array $options = [
    'plugins' => [
        'legend' => [
            'display' => false,
        ],
    ],
];
```

Alternatively, you can override the `getOptions()` method to return a dynamic array of options:

```
protected function getOptions(): array
{
    return [
        'plugins' => [
            'legend' => [
                'display' => false,
            ],
        ],
    ];
}
```

These PHP arrays will get transformed into JSON objects when the chart is rendered. If you want to return raw JavaScript from this method instead, you can return a `RawJs` object. This is useful if you want to use a JavaScript callback function, for example:

```

use Filament\Support\RawJs;

protected function getOptions(): RawJs
{
    return RawJs::make(<<<JS
        {
            scales: {
                y: {
                    ticks: {
                        callback: (value) => '€' + value,
                    },
                },
            }
        }
    JS);
}

```

Adding a description

You may add a description, below the heading of the chart, using the `getDescription()` method:

```

public function getDescription(): ?string
{
    return 'The number of blog posts published per month.';
}

```

Disabling lazy loading

By default, widgets are lazy-loaded. This means that they will only be loaded when they are visible on the page.

To disable this behavior, you may override the `$isLazy` property on the widget class:

```
protected static bool $isLazy = true;
```

Using custom Chart.js plugins

Chart.js offers a powerful plugin system that allows you to extend its functionality and create custom chart behaviors. This guide details how to use them in a chart widget.

Step 1: Install the plugin with NPM

To start with, install the plugin using NPM into your project. In this guide, we will install `chartjs-plugin-datalabels`:

```
npm install chartjs-plugin-datalabels --save-dev
```

Step 2: Create a JavaScript file importing the plugin

Create a new JavaScript file where you will define your custom plugin. In this guide, we'll call it `filament-chart-js-plugins.js`. Import the plugin, and add it to the `window.filamentChartJsPlugins` array:

```
import ChartDataLabels from 'chartjs-plugin-datalabels'

window.filamentChartJsPlugins ??= []
window.filamentChartJsPlugins.push(ChartDataLabels)
```

It's important to initialise the array if it has not been already, before pushing onto it. This ensures that multiple JavaScript files (especially those from Filament plugins) that register Chart.js plugins do not overwrite each other, regardless of the order they are booted in.

You can push as many plugins to the `filamentChartJsPlugins` array as you would like to install, you do not need a separate file to import each plugin.

Step 3: Compile the JavaScript file with Vite

Now, you need to build the JavaScript file with Vite, or your bundler of choice. Include the file in your Vite configuration (usually `vite.config.js`). For example:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      input: [
        'resources/css/app.css',
        'resources/js/app.js',
        'resources/css/filament/admin/theme.css',
        'resources/js/filament-chart-js-plugins.js', // Include the new file in the
      'input` array so it is built
      1,
    },
  ],
});
```

Build the file with `npm run build`.

Step 4: Register the JavaScript file in Filament

Filament needs to know to include this JavaScript file when rendering chart widgets. You can do this in the `boot()` method of a service provider like `AppServiceProvider`:

```
use Filament\Support\Assets\Js;
use Filament\Support\Facades\FilamentAsset;
use Illuminate\Support\Facades\Vite;

FilamentAsset::register([
  Js::make('chart-js-plugins', Vite::asset('resources/js/filament-chart-js-plugins.js'))-
>module(),
]);
```

You can find out more about [asset registration](#), and even [register assets for a specific panel](#).

Tables

When using the [Panel Builder](#), you can use table widgets. These use the [table builder](#). You can find out how to create them [here](#).

If you're not using the Panel Builder, there's no need to use a "widget" to render a table. You can simply [add a table to a Livewire component](#), which does not provide any specific benefits over a widget.

Adding A Widget To A Blade View

Overview

Since widgets are Livewire components, you can easily render a widget in any Blade view using the `@livewire` directive:

```
<div>
    @livewire (\App\Livewire\Dashboard\PostsChart::class)
</div>
```

Chapter 8

Core Concepts

Overview

This section of the documentation contains information that applies to all packages in the Filament ecosystem.

Eloquent Models

All of Filament's database interactions rely on Eloquent. If your application needs to work with a static data source like a plain PHP array, you may find [Sushi](#) useful for accessing that data from an Eloquent model.

Assets

Overview

All packages in the Filament ecosystem share an asset management system. This allows both official plugins and third-party plugins to register CSS and JavaScript files that can then be consumed by Blade views.

The `FilamentAsset` facade

The `FilamentAsset` facade is used to register files into the asset system. These files may be sourced from anywhere in the filesystem, but are then copied into the `/public` directory of the application when the `php artisan filament:assets` command is run. By copying them into the `/public` directory for you, we can predictably load them in Blade views, and also ensure that third party packages are able to load their assets without having to worry about where they are located.

Assets always have a unique ID chosen by you, which is used as the file name when the asset is copied into the `/public` directory. This ID is also used to reference the asset in Blade views. While the ID is unique, if you are registering assets for a plugin, then you do not need to worry about IDs clashing with other plugins, since the asset will be copied into a directory named after your plugin.

The `FilamentAsset` facade should be used in the `boot()` method of a service provider. It can be used inside an application service provider such as `AppServiceProvider`, or inside a plugin service provider.

The `FilamentAsset` facade has one main method, `register()`, which accepts an array of assets to register:

```
use Filament\Support\Facades\FilamentAsset;

public function boot(): void
{
    // ...

    FilamentAsset::register([
        // ...
    ]);

    // ...
}
```

Registering assets for a plugin

When registering assets for a plugin, you should pass the name of the Composer package as the second argument of the `register()` method:

```
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    // ...
], package: 'danharrin/filament-blog');
```

Now, all the assets for this plugin will be copied into their own directory inside `/public`, to avoid the possibility of clashing with other plugins' files with the same names.

Registering CSS files

To register a CSS file with the asset system, use the `FilamentAsset::register()` method in the `boot()` method of a service provider. You must pass in an array of `Css` objects, which each represents a CSS file that should be registered in the asset system.

Each `Css` object has a unique ID and a path to the CSS file:

```
use Filament\Support\Assets\Css;
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    Css::make('custom-stylesheet', __DIR__ . '/../../resources/css/custom.css'),
]);
```

In this example, we use `__DIR__` to generate a relative path to the asset from the current file. For instance, if you were adding this code to `/app/Providers/AppServiceProvider.php`, then the CSS file should exist in `/resources/css/custom.css`.

Now, when the `php artisan filament:assets` command is run, this CSS file is copied into the `/public` directory. In addition, it is now loaded into all Blade views that use Filament. If you're interested in only loading the CSS when it is required by an element on the page, check out the [Lazy loading CSS](#) section.

Using Tailwind CSS in plugins

Typically, registering CSS files is used to register custom stylesheets for your application. If you want to process these files using Tailwind CSS, you need to consider the implications of that, especially if you are a plugin developer.

Tailwind builds are unique to every application - they contain a minimal set of utility classes, only the ones that you are actually using in your application. This means that if you are a plugin developer, you probably should not be building your Tailwind CSS files into your plugin. Instead, you should provide the raw CSS files and instruct the user that they should build the Tailwind CSS file themselves. To do this, they probably just need to add your vendor directory into the `content` array of their `tailwind.config.js` file:

```
export default {
    content: [
        './resources/**/*.blade.php',
        './vendor/filament/**/*.blade.php',
        './vendor/danharrin/filament-blog/resources/views/**/*.blade.php', // Your plugin's
        vendor directory
    ],
    // ...
}
```

This means that when they build their Tailwind CSS file, it will include all the utility classes that are used in your plugin's views, as well as the utility classes that are used in their application and the Filament core.

However, with this technique, there might be extra complications for users who use your plugin with the [Panel Builder](#). If they have a [custom theme](#), they will be fine, since they are building their own CSS file anyway using Tailwind CSS.

However, if they are using the default stylesheet which is shipped with the Panel Builder, you might have to be careful about the utility classes that you use in your plugin's views. For instance, if you use a utility class that is not included in the default stylesheet, the user is not compiling it themselves, and it will not be included in the final CSS file. This means that your plugin's views might not look as expected. This is one of the few situations where I would recommend compiling and [registering](#) a Tailwind CSS-compiled stylesheet in your plugin.

Lazy loading CSS

By default, all CSS files registered with the asset system are loaded in the `<head>` of every Filament page. This is the simplest way to load CSS files, but sometimes they may be quite heavy and not required on every page. In this case, you can leverage the [Alpine.js Lazy Load Assets](#) package that comes bundled with Filament. It allows you to easily load CSS files on-demand using Alpine.js. The premise is very simple, you use the `x-load-css` directive on an element, and when that element is loaded onto the page, the specified CSS files are loaded into the `<head>` of the page. This is perfect for both small UI elements and entire pages that require a CSS file:

```
<div
  x-data="{}"
  x-load-css="@js(\Filament\Support\Facades\FilamentAsset::getStyleHref('custom-
stylesheet'))]"
>
<!-- ... -->
</div>
```

To prevent the CSS file from being loaded automatically, you can use the `loadedOnRequest()` method:

```
use Filament\Support\Assets\Css;
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    Css::make('custom-stylesheet', __DIR__ . '/../../resources/css/custom.css')-
    >loadedOnRequest(),
]);
```

If your CSS file was [registered to a plugin](#), you must pass that in as the second argument to the `FilamentAsset::getStyleHref()` method:

```
<div
  x-data="{}"
  x-load-css="@js(\Filament\Support\Facades\FilamentAsset::getStyleHref('custom-stylesheet',
package: 'danharrin/filament-blog'))"
>
<!-- ... -->
</div>
```

Registering CSS files from a URL

If you want to register a CSS file from a URL, you may do so. These assets will be loaded on every page as normal, but not copied into the `/public` directory when the `php artisan filament:assets` command is run. This is useful for registering external stylesheets from a CDN, or stylesheets that you are already compiling directly into the `/public` directory:

```
use Filament\Support\Assets\Css;
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    Css::make('example-external-stylesheet', 'https://example.com/external.css'),
    Css::make('example-local-stylesheet', asset('css/local.css')),
]);
```

Registering CSS variables

Sometimes, you may wish to use dynamic data from the backend in CSS files. To do this, you can use the `FilamentAsset::registerCssVariables()` method in the `boot()` method of a service provider:

```
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::registerCssVariables([
    'background-image' => asset('images/background.jpg'),
]);
```

Now, you can access these variables from any CSS file:

```
background-image: var(--background-image);
```

Registering JavaScript files

To register a JavaScript file with the asset system, use the `FilamentAsset::register()` method in the `boot()` method of a service provider. You must pass in an array of `Js` objects, which each represents a JavaScript file that should be registered in the asset system.

Each `Js` object has a unique ID and a path to the JavaScript file:

```
use Filament\Support\Assets\Js;

FilamentAsset::register([
    Js::make('custom-script', __DIR__ . '/../../resources/js/custom.js'),
]);
```

In this example, we use `__DIR__` to generate a relative path to the asset from the current file. For instance, if you were adding this code to `/app/Providers/AppServiceProvider.php`, then the JavaScript file should exist in `/resources/js/custom.js`.

Now, when the `php artisan filament:assets` command is run, this JavaScript file is copied into the `/public` directory. In addition, it is now loaded into all Blade views that use Filament. If you're interested in only loading the JavaScript when it is required by an element on the page, check out the [Lazy loading JavaScript](#) section.

Lazy loading JavaScript

By default, all JavaScript files registered with the asset system are loaded at the bottom of every Filament page. This is the simplest way to load JavaScript files, but sometimes they may be quite heavy and not required on every page. In this case, you can leverage the [Alpine.js Lazy Load Assets](#) package that comes bundled with Filament. It allows you to easily load JavaScript files on-demand using Alpine.js. The premise is very simple, you use the `x-load-js` directive on an element, and when that element is loaded onto the page, the specified JavaScript files are loaded at the bottom of the page. This is perfect for both small UI elements and entire pages that require a JavaScript file:

```
<div
    x-data="{}"
    x-load-js="@js(\Filament\Support\Facades\FilamentAsset::getScriptSrc('custom-script'))"
>
    <!-- ... -->
</div>
```

To prevent the JavaScript file from being loaded automatically, you can use the `loadedOnRequest()` method:

```
use Filament\Support\Assets\Js;
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    Js::make('custom-script', __DIR__ . '/../resources/js/custom.js')->loadedOnRequest(),
]);

```

If your JavaScript file was [registered to a plugin](#), you must pass that in as the second argument to the `FilamentAsset::getScriptSrc()` method:

```
<div
    x-data="{}"
    x-load-js="@js(\Filament\Support\Facades\FilamentAsset::getScriptSrc('custom-script',
    package: 'danharrin/filament-blog'))"
>
    <!-- ... -->
</div>
```

Asynchronous Alpine.js components

Sometimes, you may want to load external JavaScript libraries for your Alpine.js-based components. The best way to do this is by storing the compiled JavaScript and Alpine component in a separate file, and letting us load it whenever the component is rendered.

Firstly, you should install [esbuild](#) via NPM, which we will use to create a single JavaScript file containing your external library and Alpine component:

```
npm install esbuild --save-dev
```

Then, you must create a script to compile your JavaScript and Alpine component. You can put this anywhere, for example `bin/build.js`:

```

import * as esbuild from 'esbuild'

const isDev = process.argv.includes('--dev')

async function compile(options) {
    const context = await esbuild.context(options)

    if (isDev) {
        await context.watch()
    } else {
        await context.rebuild()
        await context.dispose()
    }
}

const defaultOptions = {
    define: {
        'process.env.NODE_ENV': isDev ? `development` : `production`,
    },
    bundle: true,
    mainFields: ['module', 'main'],
    platform: 'neutral',
    sourcemap: isDev ? 'inline' : false,
    sourcesContent: isDev,
    treeShaking: true,
    target: ['es2020'],
    minify: !isDev,
    plugins: [
        {
            name: 'watchPlugin',
            setup: function (build) {
                build.onStart(() => {
                    console.log(`Build started at ${new Date(Date.now()).toLocaleTimeString()}: ${build.initialOptions.outfile}`)
                })

                build.onEnd((result) => {
                    if (result.errors.length > 0) {
                        console.log(`Build failed at ${new Date(Date.now()).toLocaleTimeString()}: ${build.initialOptions.outfile}`, result.errors)
                    } else {
                        console.log(`Build finished at ${new Date(Date.now()).toLocaleTimeString()}: ${build.initialOptions.outfile}`)
                    }
                })
            }
        ],
    }
}

compile({
    ...defaultOptions,
    entryPoints: ['./resources/js/components/test-component.js'],
    outfile: './resources/js/dist/components/test-component.js',
})

```

As you can see at the bottom of the script, we are compiling a file called `resources/js/components/test-component.js` into `resources/js/dist/components/test-component.js`. You can change these paths to suit your needs. You can compile as many components as you want.

Now, create a new file called `resources/js/components/test-component.js`:

```
// Import any external JavaScript libraries from NPM here.

export default function testComponent({
    state,
}) {
    return {
        state,

        // You can define any other Alpine.js properties here.

        init: function () {
            // Initialise the Alpine component here, if you need to.
        },

        // You can define any other Alpine.js functions here.
    }
}
```

Now, you can compile this file into `resources/js/dist/components/test-component.js` by running the following command:

```
node bin/build.js
```

If you want to watch for changes to this file instead of compiling once, try the following command:

```
node bin/build.js --dev
```

Now, you need to tell Filament to publish this compiled JavaScript file into the `/public` directory of the Laravel application, so it is accessible to the browser. To do this, you can use the `FilamentAsset::register()` method in the `boot()` method of a service provider, passing in an `AlpineComponent` object:

```
use Filament\Support\Assets\AlpineComponent;
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::register([
    AlpineComponent::make('test-component', __DIR__ . '/../../resources/js/dist/components/test-component.js'),
]);
```

When you run `php artisan filament:assets`, the compiled file will be copied into the `/public` directory.

Finally, you can load this asynchronous Alpine component in your view using `ax-load` attributes and the `FilamentAsset::getAlpineComponentSrc()` method:

```
<div
    x-ignore
    ax-load
    ax-load-src="{!! \Filament\Support\Facades\FilamentAsset::getAlpineComponentSrc('test-
component') !!}"
    x-data="testComponent({
        state: $wire>{!! $applyStateBindingModifiers("\$entangle('{$statePath}')") !!},
    })"
>
    <input x-model="state" />
</div>
```

This example is for a [custom form field](#). It passes the `state` in as a parameter to the `testComponent()` function, which is entangled with a Livewire component property. You can pass in any parameters you want, and access them in the `testComponent()` function. If you're not using a custom form field, you can ignore the `state` parameter in this example.

The `ax-load` attributes come from the [Async Alpine](#) package, and any features of that package can be used here.

Registering script data

Sometimes, you may wish to make data from the backend available to JavaScript files. To do this, you can use the `FilamentAsset::registerScriptData()` method in the `boot()` method of a service provider:

```
use Filament\Support\Facades\FilamentAsset;

FilamentAsset::registerScriptData([
    'user' => [
        'name' => auth() -> user() ?-> name,
    ],
]);
```

Now, you can access that data from any JavaScript file at runtime, using the `window.filamentData` object:

```
window.filamentData.user.name // 'Dan Harrin'
```

Registering JavaScript files from a URL

If you want to register a JavaScript file from a URL, you may do so. These assets will be loaded on every page as normal, but not copied into the `/public` directory when the `php artisan filament:assets` command is run. This is useful for registering external scripts from a CDN, or scripts that you are already compiling directly into the `/public` directory:

```
use Filament\Support\Assets\Js;

FilamentAsset::register([
    Js::make('example-external-script', 'https://example.com/external.js'),
    Js::make('example-local-script', asset('js/local.js')),
]);
```

Icons

Overview

Icons are used throughout the entire Filament UI to visually communicate core parts of the user experience. To render icons, we use the [Blade Icons](#) package from Blade UI Kit.

They have a website where you can [search all the available icons](#) from various Blade Icons packages. Each package contains a different icon set that you can choose from.

Using custom SVGs as icons

The [Blade Icons](#) package allows you to register custom SVGs as icons. This is useful if you want to use your own custom icons in Filament.

To start with, publish the Blade Icons configuration file:

```
php artisan vendor:publish --tag=blade-icons
```

Now, open the `config/blade-icons.php` file, and uncomment the `default` set in the `sets` array.

Now that the default set exists in the config file, you can simply put any icons you want inside the `resources/svg` directory of your application. For example, if you put an SVG file named `star.svg` inside the `resources/svg` directory, you can reference it anywhere in Filament as `icon-star`. The `icon-` prefix is configurable in the `config/blade-icons.php` file too. You can also render the custom icon in a Blade view using the `@svg('icon-star')` directive.

Replacing the default icons

Filament includes an icon management system that allows you to replace any icons are used by default in the UI with your own. This happens in the `boot()` method of any service provider, like `AppServiceProvider`, or even a dedicated service provider for icons. If you wanted to build a plugin to replace Heroicons with a different set, you could absolutely do that by creating a Laravel package with a similar service provider.

To replace an icon, you can use the `FilamentIcon` facade. It has a `register()` method, which accepts an array of icons to replace. The key of the array is the unique `icon alias` that identifies the icon in the Filament UI, and the value is name of a Blade icon to replace it instead. Alternatively, you may use HTML instead of an icon name to render an icon from a Blade view for example:

```
use Filament\Support\Facades\FilamentIcon;

FilamentIcon::register([
    'panels::topbar.global-search.field' => 'fas-magnifying-glass',
    'panels::sidebar.group.collapse-button' => view('icons.chevron-up'),
]);
```

Allowing users to customize icons from your plugin

If you have built a Filament plugin, your users may want to be able to customize icons in the same way that they can with any core Filament package. This is possible if you replace any manual `@svg()` usages with the `<x-filament::icon>` Blade component. This component allows you to pass in an icon alias, the name of the SVG icon that should be used by default, and any classes or HTML attributes:

```
<x-filament::icon
    alias="panels::topbar.global-search.field"
    icon="heroicon-m-magnifying-glass"
    wire:target="search"
    class="h-5 w-5 text-gray-500 dark:text-gray-400"
/>
```

Alternatively, you may pass an SVG element into the component's slot instead of defining a default icon name:

```
<x-filament::icon
    alias="panels::topbar.global-search.field"
    wire:target="search"
    class="h-5 w-5 text-gray-500 dark:text-gray-400"
>
<svg>
    <!-- ... -->
</svg>
</x-filament::icon>
```

Available icon aliases

Panel Builder icon aliases

- `panels::global-search.field` - Global search field
- `panels::pages.dashboard.actions.filter` - Trigger button of the dashboard filter action
- `panels::pages.dashboard.navigation-item` - Dashboard page navigation item
- `panels::pages.password-reset.request-password-reset.actions.login` - Trigger button of the login action on the request password reset page
- `panels::pages.password-reset.request-password-reset.actions.login rtl` - Trigger button of the login action on the request password reset page (right-to-left direction)
- `panels::resources.pages.edit-record.navigation-item` - Resource edit record page navigation item
- `panels::resources.pages.manage-related-records.navigation-item` - Resource manage related records page navigation item
- `panels::resources.pages.view-record.navigation-item` - Resource view record page navigation item
- `panels::sidebar.collapse-button` - Button to collapse the sidebar
- `panels::sidebar.collapse-button.rtl` - Button to collapse the sidebar (right-to-left direction)
- `panels::sidebar.expand-button` - Button to expand the sidebar
- `panels::sidebar.expand-button.rtl` - Button to expand the sidebar (right-to-left direction)
- `panels::sidebar.group.collapse-button` - Collapse button for a sidebar group
- `panels::tenant-menu.billing-button` - Billing button in the tenant menu
- `panels::tenant-menu.profile-button` - Profile button in the tenant menu
- `panels::tenant-menu.registration-button` - Registration button in the tenant menu
- `panels::tenant-menu.toggle-button` - Button to toggle the tenant menu
- `panels::theme-switcher.light-button` - Button to switch to the light theme from the theme switcher
- `panels::theme-switcher.dark-button` - Button to switch to the dark theme from the theme switcher
- `panels::theme-switcher.system-button` - Button to switch to the system theme from the theme switcher
- `panels::topbar.close-sidebar-button` - Button to close the sidebar
- `panels::topbar.open-sidebar-button` - Button to open the sidebar
- `panels::topbar.group.toggle-button` - Toggle button for a topbar group
- `panels::topbar.open-database-notifications-button` - Button to open the database notifications modal
- `panels::user-menu.profile-item` - Profile item in the user menu
- `panels::user-menu.logout-button` - Button in the user menu to log out

- `panels::widgets.account.logout-button` - Button in the account widget to log out
- `panels::widgets.filament-info.open-documentation-button` - Button to open the documentation from the Filament info widget
- `panels::widgets.filament-info.open-github-button` - Button to open GitHub from the Filament info widget

Form Builder icon aliases

- `forms::components.builder.actions.clone` - Trigger button of a clone action in a builder item
- `forms::components.builder.actions.collapse` - Trigger button of a collapse action in a builder item
- `forms::components.builder.actions.delete` - Trigger button of a delete action in a builder item
- `forms::components.builder.actions.expand` - Trigger button of an expand action in a builder item
- `forms::components.builder.actions.move-down` - Trigger button of a move down action in a builder item
- `forms::components.builder.actions.move-up` - Trigger button of a move up action in a builder item
- `forms::components.builder.actions.reorder` - Trigger button of a reorder action in a builder item
- `forms::components.checkbox-list.search-field` - Search input in a checkbox list
- `forms::components.file-upload.editor.actions.drag-crop` - Trigger button of a drag crop action in a file upload editor
- `forms::components.file-upload.editor.actions.drag-move` - Trigger button of a drag move action in a file upload editor
- `forms::components.file-upload.editor.actions.flip-horizontal` - Trigger button of a flip horizontal action in a file upload editor
- `forms::components.file-upload.editor.actions.flip-vertical` - Trigger button of a flip vertical action in a file upload editor
- `forms::components.file-upload.editor.actions.move-down` - Trigger button of a move down action in a file upload editor
- `forms::components.file-upload.editor.actions.move-left` - Trigger button of a move left action in a file upload editor
- `forms::components.file-upload.editor.actions.move-right` - Trigger button of a move right action in a file upload editor
- `forms::components.file-upload.editor.actions.move-up` - Trigger button of a move up action in a file upload editor
- `forms::components.file-upload.editor.actions.rotate-left` - Trigger button of a rotate left action in a file upload editor
- `forms::components.file-upload.editor.actions.rotate-right` - Trigger button of a rotate right action in a file upload editor
- `forms::components.file-upload.editor.actions.zoom-100` - Trigger button of a zoom 100 action in a file upload editor
- `forms::components.file-upload.editor.actions.zoom-in` - Trigger button of a zoom in action in a file upload editor
- `forms::components.file-upload.editor.actions.zoom-out` - Trigger button of a zoom out action in a file upload editor
- `forms::components.key-value.actions.delete` - Trigger button of a delete action in a key-value field item
- `forms::components.key-value.actions.reorder` - Trigger button of a reorder action in a key-value field item
- `forms::components.repeater.actions.clone` - Trigger button of a clone action in a repeater item
- `forms::components.repeater.actions.collapse` - Trigger button of a collapse action in a repeater item
- `forms::components.repeater.actions.delete` - Trigger button of a delete action in a repeater item
- `forms::components.repeater.actions.expand` - Trigger button of an expand action in a repeater item
- `forms::components.repeater.actions.move-down` - Trigger button of a move down action in a repeater item
- `forms::components.repeater.actions.move-up` - Trigger button of a move up action in a repeater item
- `forms::components.repeater.actions.reorder` - Trigger button of a reorder action in a repeater item

- `forms::components.select.actions.create-option` - Trigger button of a create option action in a select field
- `forms::components.select.actions.edit-option` - Trigger button of an edit option action in a select field
- `forms::components.text-input.actions.hide-password` - Trigger button of a hide password action in a text input field
- `forms::components.text-input.actions.show-password` - Trigger button of a show password action in a text input field
- `forms::components.toggle-buttons.boolean.false` - "False" option of a `boolean()` toggle buttons field
- `forms::components.toggle-buttons.boolean.true` - "True" option of a `boolean()` toggle buttons field
- `forms::components.wizard.completed-step` - Completed step in a wizard

Table Builder icon aliases

- `tables::actions.disable-reordering` - Trigger button of the disable reordering action
- `tables::actions.enable-reordering` - Trigger button of the enable reordering action
- `tables::actions.filter` - Trigger button of the filter action
- `tables::actions.group` - Trigger button of a group records action
- `tables::actions.open-bulk-actions` - Trigger button of an open bulk actions action
- `tables::actions.toggle-columns` - Trigger button of the toggle columns action
- `tables::columns.collapse-button` - Button to collapse a column
- `tables::columns.icon-column.false` - Falsy state of an icon column
- `tables::columns.icon-column.true` - Truthy state of an icon column
- `tables::empty-state` - Empty state icon
- `tables::filters.query-builder.constraints.boolean` - Default icon for a boolean constraint in the query builder
- `tables::filters.query-builder.constraints.date` - Default icon for a date constraint in the query builder
- `tables::filters.query-builder.constraints.number` - Default icon for a number constraint in the query builder
- `tables::filters.query-builder.constraints.relationship` - Default icon for a relationship constraint in the query builder
- `tables::filters.query-builder.constraints.select` - Default icon for a select constraint in the query builder
- `tables::filters.query-builder.constraints.text` - Default icon for a text constraint in the query builder
- `tables::filters.remove-all-button` - Button to remove all filters
- `tables::grouping.collapse-button` - Button to collapse a group of records
- `tables::header-cell.sort-asc-button` - Sort button of a column sorted in ascending order
- `tables::header-cell.sort-desc-button` - Sort button of a column sorted in descending order
- `tables::reorder.handle` - Handle to grab in order to reorder a record with drag and drop
- `tables::search-field` - Search input

Notifications icon aliases

- `notifications::database.modal.empty-state` - Empty state of the database notifications modal
- `notifications::notification.close-button` - Button to close a notification
- `notifications::notification.danger` - Danger notification
- `notifications::notification.info` - Info notification
- `notifications::notification.success` - Success notification
- `notifications::notification.warning` - Warning notification

Actions icon aliases

- `actions::action-group` - Trigger button of an action group
- `actions::create-action.grouped` - Trigger button of a grouped create action
- `actions::delete-action` - Trigger button of a delete action

- `actions::delete-action.grouped` - Trigger button of a grouped delete action
- `actions::delete-action.modal` - Modal of a delete action
- `actions::detach-action` - Trigger button of a detach action
- `actions::detach-action.modal` - Modal of a detach action
- `actions::dissociate-action` - Trigger button of a dissociate action
- `actions::dissociate-action.modal` - Modal of a dissociate action
- `actions::edit-action` - Trigger button of an edit action
- `actions::edit-action.grouped` - Trigger button of a grouped edit action
- `actions::export-action.grouped` - Trigger button of a grouped export action
- `actions::force-delete-action` - Trigger button of a force delete action
- `actions::force-delete-action.grouped` - Trigger button of a grouped force delete action
- `actions::force-delete-action.modal` - Modal of a force delete action
- `actions::import-action.grouped` - Trigger button of a grouped import action
- `actions::modal.confirmation` - Modal of an action that requires confirmation
- `actions::replicate-action` - Trigger button of a replicate action
- `actions::replicate-action.grouped` - Trigger button of a grouped replicate action
- `actions::restore-action` - Trigger button of a restore action
- `actions::restore-action.grouped` - Trigger button of a grouped restore action
- `actions::restore-action.modal` - Modal of a restore action
- `actions::view-action` - Trigger button of a view action
- `actions::view-action.grouped` - Trigger button of a grouped view action

Infolist Builder icon aliases

- `infolists::components.icon-entry.false` - Falsy state of an icon entry
- `infolists::components.icon-entry.true` - Truthy state of an icon entry

UI components icon aliases

- `badge.delete-button` - Button to delete a badge
- `breadcrumbs.separator` - Separator between breadcrumbs
- `breadcrumbs.separator.rtl` - Separator between breadcrumbs (right-to-left direction)
- `modal.close-button` - Button to close a modal
- `pagination.first-button` - Button to go to the first page
- `pagination.first-button.rtl` - Button to go to the first page (right-to-left direction)
- `pagination.last-button` - Button to go to the last page
- `pagination.last-button.rtl` - Button to go to the last page (right-to-left direction)
- `pagination.next-button` - Button to go to the next page
- `pagination.next-button.rtl` - Button to go to the next page (right-to-left direction)
- `pagination.previous-button` - Button to go to the previous page
- `pagination.previous-button.rtl` - Button to go to the previous page (right-to-left direction)
- `section.collapse-button` - Button to collapse a section

Colors

Overview

Filament uses CSS variables to define its color palette. These CSS variables are mapped to Tailwind classes in the preset file that you load when installing Filament.

Customizing the default colors

From a service provider's `boot()` method, or middleware, you can call the `FilamentColor::register()` method, which you can use to customize which colors Filament uses for UI elements.

There are 6 default colors that are used throughout Filament that you are able to customize:

```
use Filament\Support\Colors\Color;
use Filament\Support\Facades\FilamentColor;

FilamentColor::register([
    'danger' => Color::Red,
    'gray' => Color::Zinc,
    'info' => Color::Blue,
    'primary' => Color::Amber,
    'success' => Color::Green,
    'warning' => Color::Amber,
]);
```

The `Color` class contains every [Tailwind CSS color](#) to choose from.

You can also pass in a function to `register()` which will only get called when the app is getting rendered. This is useful if you are calling `register()` from a service provider, and want to access objects like the currently authenticated user, which are initialized later in middleware.

Using a non-Tailwind color

You can use custom colors that are not included in the [Tailwind CSS color](#) palette by passing an array of color shades from `50` to `950` in RGB format:

```
use Filament\Support\Facades\FilamentColor;

FilamentColor::register([
    'danger' => [
        50 => '254, 242, 242',
        100 => '254, 226, 226',
        200 => '254, 202, 202',
        300 => '252, 165, 165',
        400 => '248, 113, 113',
        500 => '239, 68, 68',
        600 => '220, 38, 38',
        700 => '185, 28, 28',
        800 => '153, 27, 27',
        900 => '127, 29, 29',
        950 => '69, 10, 10',
    ],
]);
```

Generating a custom color from a hex code

You can use the `Color::hex()` method to generate a custom color palette from a hex code:

```
use Filament\Support\Colors\Color;
use Filament\Support\Facades\FilamentColor;

FilamentColor::register([
    'danger' => Color::hex('#ff0000'),
]);
```

Generating a custom color from an RGB value

You can use the `Color::rgb()` method to generate a custom color palette from an RGB value:

```
use Filament\Support\Colors\Color;
use Filament\Support\Facades\FilamentColor;

FilamentColor::register([
    'danger' => Color::rgb('rgb(255, 0, 0)'),
]);
```

Registering extra colors

You can register extra colors that you can use throughout Filament:

```
use Filament\Support\Colors\Color;
use Filament\Support\Facades\FilamentColor;

FilamentColor::register([
    'indigo' => Color::Indigo,
]);
```

Now, you can use this color anywhere you would normally add `primary`, `danger`, etc.

Style Customization

Overview

Filament uses CSS "hook" classes to allow various HTML elements to be customized using CSS.

Discovering hook classes

We could document all the hook classes across the entire Filament UI, but that would be a lot of work, and probably not very useful to you. Instead, we recommend using your browser's developer tools to inspect the elements you want to customize, and then use the hook classes to target those elements.

All hook classes are prefixed with `fi-`, which is a great way to identify them. They are usually right at the start of the class list, so they are easy to find, but sometimes they may fall further down the list if we have to apply them conditionally with JavaScript or Blade.

If you don't find a hook class you're looking for, try not to hack around it, as it might expose your styling customizations to breaking changes in future releases. Instead, please open a pull request to add the hook class you need. We can help you maintain naming consistency. You probably don't even need to pull down the Filament repository locally for these pull requests, as you can just edit the Blade files directly on GitHub.

Applying styles to hook classes

For example, if you want to customize the color of the sidebar, you can inspect the sidebar element in your browser's developer tools, see that it uses the `fi-sidebar`, and then add CSS to your app like this:

```
.fi-sidebar {
    background-color: #fafa;
}
```

Alternatively, since Filament is built upon Tailwind CSS, you can use their `@apply` directive to apply Tailwind classes to Filament elements:

```
.fi-sidebar {
    @apply bg-gray-50 dark:bg-gray-950;
}
```

Occasionally, you may need to use the `!important` modifier to override existing styles, but please use this sparingly, as it can make your styles difficult to maintain:

```
.fi-sidebar {
    @apply bg-gray-50 dark:bg-gray-950 !important;
}
```

You can even apply `!important` to only specific Tailwind classes, which is a little less intrusive, by prefixing the class name with `!`:

```
.fi-sidebar {
    @apply !bg-gray-50 dark:!bg-gray-950;
}
```

Common hook class abbreviations

We use a few common abbreviations in our hook classes to keep them short and readable:

- `fi` is short for "Filament"
- `fi-ac` is used to represent classes used in the Actions package
- `fi-fo` is used to represent classes used in the Form Builder package
- `fi-in` is used to represent classes used in the Infolist Builder package
- `fi-no` is used to represent classes used in the Notifications package
- `fi-ta` is used to represent classes used in the Table Builder package
- `fi-wi` is used to represent classes used in the Widgets package
- `btn` is short for "button"
- `col` is short for "column"
- `ctn` is short for "container"
- `wrp` is short for "wrapper"

Publishing Blade views

You may be tempted to publish the internal Blade views to your application so that you can customize them. We don't recommend this, as it will introduce breaking changes into your application in future updates. Please use the [CSS hook classes](#) wherever possible.

If you do decide to publish the Blade views, please lock all Filament packages to a specific version in your `composer.json` file, and then update Filament manually by bumping this number, testing your entire application after each update. This will help you identify breaking changes safely.

Render Hooks

Overview

Filament allows you to render Blade content at various points in the frameworks views. It's useful for plugins to be able to inject HTML into the framework. Also, since Filament does not recommend publishing the views due to an increased risk of breaking changes, it's also useful for users.

Registering render hooks

To register render hooks, you can call `FilamentView::registerRenderHook()` from a service provider or middleware. The first argument is the name of the render hook, and the second argument is a callback that returns the content to be rendered:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;
use Illuminate\Support\Facades\Blade;

FilamentView::registerRenderHook(
    PanelsRenderHook::BODY_START,
    fn (): string => Blade::render('@livewire(\'livewire-ui-modal\')'),
);
```

You could also render view content from a file:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;
use Illuminate\Contracts\View\View;

FilamentView::registerRenderHook(
    PanelsRenderHook::BODY_START,
    fn (): View => view('impersonation-banner'),
);
```

Available render hooks

Panel Builder render hooks

```
use Filament\View\PanelsRenderHook;
```

- `PanelsRenderHook::AUTH_LOGIN_FORM_AFTER` - After login form
- `PanelsRenderHook::AUTH_LOGIN_FORM_BEFORE` - Before login form
- `PanelsRenderHook::AUTH_PASSWORD_RESET_REQUEST_FORM_AFTER` - After password reset request form
- `PanelsRenderHook::AUTH_PASSWORD_RESET_REQUEST_FORM_BEFORE` - Before password reset request form
- `PanelsRenderHook::AUTH_PASSWORD_RESET_RESET_FORM_AFTER` - After password reset form
- `PanelsRenderHook::AUTH_PASSWORD_RESET_RESET_FORM_BEFORE` - Before password reset form
- `PanelsRenderHook::AUTH_REGISTER_FORM_AFTER` - After register form
- `PanelsRenderHook::AUTH_REGISTER_FORM_BEFORE` - Before register form
- `PanelsRenderHook::BODY_END` - Before `</body>`

- `PanelsRenderHook::BODY_START` - After `<body>`
- `PanelsRenderHook::CONTENT_END` - After page content, inside `<main>`
- `PanelsRenderHook::CONTENT_START` - Before page content, inside `<main>`
- `PanelsRenderHook::FOOTER` - Footer of the page
- `PanelsRenderHook::GLOBAL_SEARCH_AFTER` - After the `global search` container, inside the topbar
- `PanelsRenderHook::GLOBAL_SEARCH_BEFORE` - Before the `global search` container, inside the topbar
- `PanelsRenderHook::GLOBAL_SEARCH_END` - The end of the `global search` container
- `PanelsRenderHook::GLOBAL_SEARCH_START` - The start of the `global search` container
- `PanelsRenderHook::HEAD_END` - Before `</head>`
- `PanelsRenderHook::HEAD_START` - After `<head>`
- `PanelsRenderHook::PAGE_END` - End of the page content container, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_FOOTER_WIDGETS_AFTER` - After the page footer widgets, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_FOOTER_WIDGETS_BEFORE` - Before the page footer widgets, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_HEADER_ACTIONS_AFTER` - After the page header actions, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_HEADER_ACTIONS_BEFORE` - Before the page header actions, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_HEADER_WIDGETS_AFTER` - After the page header widgets, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_HEADER_WIDGETS_BEFORE` - Before the page header widgets, also can be scoped to the page or resource class
- `PanelsRenderHook::PAGE_START` - Start of the page content container, also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_LIST_RECORDS_TABLE_AFTER` - After the resource table, also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_LIST_RECORDS_TABLE_BEFORE` - Before the resource table, also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_LIST_RECORDS_TABS_END` - The end of the filter tabs (after the last tab), also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_LIST_RECORDS_TABS_START` - The start of the filter tabs (before the first tab), also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_MANAGE RELATED_RECORDS_TABLE_AFTER` - After the relation manager table, also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_PAGES_MANAGE RELATED_RECORDS_TABLE_BEFORE` - Before the relation manager table, also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_RELATION_MANAGER_AFTER` - After the relation manager table, also can be scoped to the page or relation manager class
- `PanelsRenderHook::RESOURCE_RELATION_MANAGER_BEFORE` - Before the relation manager table, also can be scoped to the page or relation manager class
- `PanelsRenderHook::RESOURCE_TABS_END` - The end of the resource tabs (after the last tab), also can be scoped to the page or resource class
- `PanelsRenderHook::RESOURCE_TABS_START` - The start of the resource tabs (before the first tab), also can be scoped to the page or resource class
- `PanelsRenderHook::SCRIPTS_AFTER` - After scripts are defined
- `PanelsRenderHook::SCRIPTS_BEFORE` - Before scripts are defined
- `PanelsRenderHook::SIDEBAR_NAV_END` - In the `sidebar`, before `</nav>`
- `PanelsRenderHook::SIDEBAR_NAV_START` - In the `sidebar`, after `<nav>`
- `PanelsRenderHook::SIMPLE_PAGE_END` - End of the simple page content container, also can be scoped to the page class

- `PanelsRenderHook::SIMPLE_PAGE_START` - Start of the simple page content container, also can be scoped to the page class
- `PanelsRenderHook::SIDEBAR_FOOTER` - Pinned to the bottom of the sidebar, below the content
- `PanelsRenderHook::STYLES_AFTER` - After styles are defined
- `PanelsRenderHook::STYLES_BEFORE` - Before styles are defined
- `PanelsRenderHook::TENANT_MENU_AFTER` - After the tenant menu
- `PanelsRenderHook::TENANT_MENU_BEFORE` - Before the tenant menu
- `PanelsRenderHook::TOPBAR_AFTER` - Below the topbar
- `PanelsRenderHook::TOPBAR_BEFORE` - Above the topbar
- `PanelsRenderHook::TOPBAR_END` - End of the topbar container
- `PanelsRenderHook::TOPBAR_START` - Start of the topbar container
- `PanelsRenderHook::USER_MENU_AFTER` - After the user menu
- `PanelsRenderHook::USER_MENU_BEFORE` - Before the user menu
- `PanelsRenderHook::USER_MENU_PROFILE_AFTER` - After the profile item in the user menu
- `PanelsRenderHook::USER_MENU_PROFILE_BEFORE` - Before the profile item in the user menu

Table Builder render hooks

All these render hooks can be scoped to any table Livewire component class. When using the Panel Builder, these classes might be the List or Manage page of a resource, or a relation manager. Table widgets are also Livewire component classes.

```
use Filament\Tables\View\TablesRenderHook;
```

- `TablesRenderHook::SELECTION_INDICATOR_ACTIONS_AFTER` - After the "select all" and "deselect all" action buttons in the selection indicator bar
- `TablesRenderHook::SELECTION_INDICATOR_ACTIONS_BEFORE` - Before the "select all" and "deselect all" action buttons in the selection indicator bar
- `TablesRenderHook::HEADER_AFTER` - After the header container
- `TablesRenderHook::HEADER_BEFORE` - Before the header container
- `TablesRenderHook::TOOLBAR_AFTER` - After the toolbar container
- `TablesRenderHook::TOOLBAR_BEFORE` - Before the toolbar container
- `TablesRenderHook::TOOLBAR_END` - The end of the toolbar
- `TablesRenderHook::TOOLBAR_GROUPING_SELECTOR_AFTER` - After the grouping selector
- `TablesRenderHook::TOOLBAR_GROUPING_SELECTOR_BEFORE` - Before the grouping selector
- `TablesRenderHook::TOOLBAR_REORDER_TRIGGER_AFTER` - After the reorder trigger
- `TablesRenderHook::TOOLBAR_REORDER_TRIGGER_BEFORE` - Before the reorder trigger
- `TablesRenderHook::TOOLBAR_SEARCH_AFTER` - After the search container
- `TablesRenderHook::TOOLBAR_SEARCH_BEFORE` - Before the search container
- `TablesRenderHook::TOOLBAR_START` - The start of the toolbar
- `TablesRenderHook::TOOLBAR_TOGGLE_COLUMN_TRIGGER_AFTER` - After the toggle columns trigger
- `TablesRenderHook::TOOLBAR_TOGGLE_COLUMN_TRIGGER_BEFORE` - Before the toggle columns trigger

Widgets render hooks

```
use Filament\widgets\View\WidgetsRenderHook;
```

- `WidgetsRenderHook::TABLE_WIDGET_END` - End of the table widget, after the table itself, also can be scoped to the table widget class
- `WidgetsRenderHook::TABLE_WIDGET_START` - Start of the table widget, before the table itself, also can be scoped to the table widget class

Scoping render hooks

Some render hooks can be given a "scope", which allows them to only be output on a specific page or Livewire component. For instance, you might want to register a render hook for just 1 page. To do that, you can pass the class of the page or component as the second argument to `registerRenderHook()`:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;
use Illuminate\Support\Facades\Blade;

FilamentView::registerRenderHook(
    PanelsRenderHook::PAGE_START,
    fn (): View => view('warning-banner'),
    scopes: \App\Filament\Resources\UserResource\Pages>EditUser::class,
);
```

You can also pass an array of scopes to register the render hook for:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;

FilamentView::registerRenderHook(
    PanelsRenderHook::PAGE_START,
    fn (): View => view('warning-banner'),
    scopes: [
        \App\Filament\Resources\UserResource\Pages>CreateUser::class,
        \App\Filament\Resources\UserResource\Pages>EditUser::class,
    ],
);
```

Some render hooks for the [Panel Builder](#) allow you to scope hooks to all pages in a resource:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;

FilamentView::registerRenderHook(
    PanelsRenderHook::PAGE_START,
    fn (): View => view('warning-banner'),
    scopes: \App\Filament\Resources\UserResource::class,
);
```

Retrieving the currently active scopes inside the render hook

The `$scopes` are passed to the render hook function, and you can use them to determine which page or component the render hook is being rendered on:

```
use Filament\Support\Facades\FilamentView;
use Filament\View\PanelsRenderHook;

FilamentView::registerRenderHook(
    PanelsRenderHook::PAGE_START,
    fn (array $scopes): View => view('warning-banner', ['scopes' => $scopes]),
    scopes: \App\Filament\Resources\UserResource::class,
);
```

Rendering hooks

Plugin developers might find it useful to expose render hooks to their users. You do not need to register them anywhere, simply output them in Blade like so:

```
{  
    \Filament\Support\Facades\FilamentView::renderHook(\Filament\View\PanelsRenderHook::PAGE_START)  
}
```

To provide scope your render hook, you can pass it as the second argument to `renderHook()`. For instance, if your hook is inside a Livewire component, you can pass the class of the component using `static::class`:

```
{  
    \Filament\Support\Facades\FilamentView::renderHook(\Filament\View\PanelsRenderHook::PAGE_START,  
    scopes: $this->getRenderHookScopes()) }
```

You can even pass multiple scopes as an array, and all render hooks that match any of the scopes will be rendered:

```
{  
    \Filament\Support\Facades\FilamentView::renderHook(\Filament\View\PanelsRenderHook::PAGE_START,  
    scopes: [static::class, \App\Filament\Resources\UserResource::class]) }
```

Enums

Overview

Enums are special PHP classes that represent a fixed set of constants. They are useful for modeling concepts that have a limited number of possible values, like days of the week, months in a year, or the suits in a deck of cards.

Since enum "cases" are instances of the enum class, adding interfaces to enums proves to be very useful. Filament provides a collection of interfaces that you can add to enums, which enhance your experience when working with them.

When using an enum with an attribute on your Eloquent model, please [ensure that it is cast correctly](#).

Enum labels

The `HasLabel` interface transforms an enum instance into a textual label. This is useful for displaying human-readable enum values in your UI.

```
use Filament\Support\Contracts\HasLabel;

enum Status: string implements HasLabel
{
    case Draft = 'draft';
    case Reviewing = 'reviewing';
    case Published = 'published';
    case Rejected = 'rejected';

    public function getLabel(): ?string
    {
        return $this->name;

        // or

        return match ($this) {
            self::Draft => 'Draft',
            self::Reviewing => 'Reviewing',
            self::Published => 'Published',
            self::Rejected => 'Rejected',
        };
    }
}
```

Using the enum label with form field options

The `HasLabel` interface can be used to generate an array of options from an enum, where the enum's value is the key and the enum's label is the value. This applies to Form Builder fields like `Select` and `CheckboxList`, as well as the Table Builder's `SelectColumn` and `SelectFilter`:

```

use Filament\Forms\Components\CheckboxList;
use Filament\Forms\Components\Radio;
use Filament\Forms\Components>Select;
use Filament\Tables\Columns\SelectColumn;
use Filament\Tables\Filters\SelectFilter;

Select::make('status')
->options(Status::class)

CheckboxList::make('status')
->options(Status::class)

Radio::make('status')
->options(Status::class)

SelectColumn::make('status')
->options(Status::class)

SelectFilter::make('status')
->options(Status::class)

```

In these examples, `Status::class` is the enum class which implements `HasLabel`, and the options are generated from that:

```
[
  'draft' => 'Draft',
  'reviewing' => 'Reviewing',
  'published' => 'Published',
  'rejected' => 'Rejected',
]
```

Using the enum label with a text column in your table

If you use a `TextColumn` with the Table Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasLabel` interface to display the enum's label instead of its raw value.

Using the enum label as a group title in your table

If you use a `grouping` with the Table Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasLabel` interface to display the enum's label instead of its raw value. The label will be displayed as the [title of each group](#).

Using the enum label with a text entry in your infolist

If you use a `TextEntry` with the Infolist Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasLabel` interface to display the enum's label instead of its raw value.

Enum colors

The `HasColor` interface transforms an enum instance into a [color](#). This is useful for displaying colored enum values in your UI.

```

use Filament\Support\Contracts\HasColor;

enum Status: string implements HasColor
{
    case Draft = 'draft';
    case Reviewing = 'reviewing';
    case Published = 'published';
    case Rejected = 'rejected';

    public function getColor(): string | array | null
    {
        return match ($this) {
            self::Draft => 'gray',
            self::Reviewing => 'warning',
            self::Published => 'success',
            self::Rejected => 'danger',
        };
    }
}

```

Using the enum color with a text column in your table

If you use a `TextColumn` with the Table Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasColor` interface to display the enum label in its color. This works best if you use the `badge()` method on the column.

Using the enum color with a text entry in your infolist

If you use a `TextEntry` with the Infolist Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasColor` interface to display the enum label in its color. This works best if you use the `badge()` method on the entry.

Using the enum color with a toggle buttons field in your form

If you use a `ToggleButtons` with the Form Builder, and it is set to use an enum for its options, Filament will automatically use the `HasColor` interface to display the enum label in its color.

Enum icons

The `HasIcon` interface transforms an enum instance into an `icon`. This is useful for displaying icons alongside enum values in your UI.

```

use Filament\Support\Contracts\HasIcon;

enum Status: string implements HasIcon
{
    case Draft = 'draft';
    case Reviewing = 'reviewing';
    case Published = 'published';
    case Rejected = 'rejected';

    public function getIcon(): ?string
    {
        return match ($this) {
            self::Draft => 'heroicon-m-pencil',
            self::Reviewing => 'heroicon-m-eye',
            self::Published => 'heroicon-m-check',
            self::Rejected => 'heroicon-m-x-mark',
        };
    }
}

```

Using the enum icon with a text column in your table

If you use a `TextColumn` with the Table Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasIcon` interface to display the enum's icon aside its label. This works best if you use the `badge()` method on the column.

Using the enum icon with a text entry in your infolist

If you use a `TextEntry` with the Infolist Builder, and it is cast to an enum in your Eloquent model, Filament will automatically use the `HasIcon` interface to display the enum's icon aside its label. This works best if you use the `badge()` method on the entry.

Using the enum icon with a toggle buttons field in your form

If you use a `ToggleButtons` with the Form Builder, and it is set to use an enum for its options, Filament will automatically use the `HasIcon` interface to display the enum's icon aside its label.

Enum descriptions

The `HasDescription` interface transforms an enum instance into a textual description, often displayed under its [label](#). This is useful for displaying human-friendly descriptions in your UI.

```

use Filament\Support\Contracts\HasDescription;
use Filament\Support\Contracts\HasLabel;

enum Status: string implements HasLabel, HasDescription
{
    case Draft = 'draft';
    case Reviewing = 'reviewing';
    case Published = 'published';
    case Rejected = 'rejected';

    public function getLabel(): ?string
    {
        return $this->name;
    }

    public function getDescription(): ?string
    {
        return match ($this) {
            self::Draft => 'This has not finished being written yet.',
            self::Reviewing => 'This is ready for a staff member to read.',
            self::Published => 'This has been approved by a staff member and is public on the
website.',
            self::Rejected => 'A staff member has decided this is not appropriate for the
website.',
        };
    }
}

```

Using the enum description with form field descriptions

The `HasDescription` interface can be used to generate an array of descriptions from an enum, where the enum's value is the key and the enum's description is the value. This applies to Form Builder fields like `Radio` and `CheckboxList`:

```

use Filament\Forms\Components\CheckboxList;
use Filament\Forms\Components\Radio;

Radio::make('status')
->options(Status::class)

CheckboxList::make('status')
->options(Status::class)

```

Contributing

Parts of this guide are taken from [Laravel's contribution guide](#), and it served as very useful inspiration.

Reporting bugs

If you find a bug in Filament, please report it by opening an issue on our [GitHub repository](#). Before opening an issue, please search the [existing issues](#) to see if the bug has already been reported.

Please make sure to include as much information as possible, including the version of packages in your app. You can use this Artisan command in your app to open a new issue with all the correct versions pre-filled:

```
php artisan make:filament-issue
```

When creating an issue, we require a "reproduction repository". **Please do not link to your actual project**, what we need instead is a *minimal* reproduction in a fresh project without any unnecessary code. This means it doesn't matter if your real project is private / confidential, since we want a link to a separate, isolated reproduction. This allows us to fix the problem much quicker. **Issues will be automatically closed and not reviewed if this is missing, to preserve maintainer time and to ensure the process is fair for those who put effort into reporting**. If you believe a reproduction repository is not suitable for the issue, which is a very rare case, please [@danharrin](#) and explain why. Saying that "it's just a simple issue" is not an excuse for not creating a repository! [Need a headstart? We have a template Filament project for you.](#)

Remember, bug reports are created in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the bug report will automatically see any activity or that others will jump to fix it. Creating a bug report serves to help yourself and others start on the path of fixing the problem.

Development of new features

If you would like to propose a new feature or improvement to Filament, you may use our [discussion form](#) hosted on GitHub. If you are intending on implementing the feature yourself in a pull request, we advise you to [@danharrin](#) in your feature discussion beforehand and ask if it is suitable for the framework to prevent wasting your time.

Development of plugins

If you would like to develop a plugin for Filament, please refer to the [plugin development section](#) here in the documentation. Our [Discord](#) server is also a great place to ask questions and get help with plugin development. You can start a conversation in the [#plugin-developers-chat](#) channel.

You can [submit your plugin to the Filament website](#).

Developing with a local copy of Filament

If you want to contribute to the Filament packages, then you may want to test it in a real Laravel project:

- Fork [the GitHub repository](#) to your GitHub account.
- Create a Laravel app locally.
- Clone your fork in your Laravel app's root directory.
- In the `/filament` directory, create a branch for your fix, e.g. `fix/error-message`.

Install the packages in your app's `composer.json`:

```
{
    // ...
    "require": {
        "filament/filament": "*",
    },
    "minimum-stability": "dev",
    "repositories": [
        {
            "type": "path",
            "url": "filament/packages/*"
        }
    ],
    // ...
}
```

Now, run `composer update`.

Once you're finished making changes, you can commit them and submit a pull request to [the GitHub repository](#).

Checking for missing translations

Set up a Laravel app, and install the [panel builder](#).

Now, if you want to check for missing Spanish translations, run:

```
php artisan filament:check-translations es
```

This will let you know which translations are missing for this locale. You can make a pull request with the changes to [the GitHub repository](#).

If you've published the translations into your app and you'd like to check those instead, try:

```
php artisan filament:check-translations es --source=app
```

Security vulnerabilities

If you discover a security vulnerability within Filament, please email Dan Harrin via dan@danharrin.com. All security vulnerabilities will be promptly addressed.

Code of Conduct

Please note that Filament is released with a [Contributor Code of Conduct](#). By participating in this project you agree to abide by its terms.

Plugins

Getting Started

Overview

While Filament comes with virtually any tool you'll need to build great apps, sometimes you'll need to add your own functionality either for just your app or as redistributable packages that other developers can include in their own apps. This is why Filament offers a plugin system that allows you to extend its functionality.

Before we dive in, it's important to understand the different contexts in which plugins can be used. There are two main contexts:

1. **Panel Plugins:** These are plugins that are used with [Panel Builders](#). They are typically used only to add functionality when used inside a Panel or as a complete Panel in and of itself. Examples of this are:
 1. A plugin that adds specific functionality to the dashboard in the form of Widgets.
 2. A plugin that adds a set of Resources / functionality to an app like a Blog or User Management feature.
2. **Standalone Plugins:** These are plugins that are used in any context outside a Panel Builder. Examples of this are:
 1. A plugin that adds custom fields to be used with the [Form Builders](#).
 2. A plugin that adds custom columns or filters to the [Table Builders](#).

Although these are two different mental contexts to keep in mind when building plugins, they can be used together inside the same plugin. They do not have to be mutually exclusive.

Important Concepts

Before we dive into the specifics of building plugins, there are a few concepts that are important to understand. You should familiarize yourself with the following before building a plugin:

1. [Laravel Package Development](#)
2. [Spatie Package Tools](#)
3. [Filament Asset Management](#)

The Plugin object

Filament v3 introduces the concept of a Plugin object that is used to configure the plugin. This object is a simple PHP class that implements the [Filament\Contracts\Plugin](#) interface. This class is used to configure the plugin and is the main entry point for the plugin. It is also used to register Resources and Icons that might be used by your plugin.

While the plugin object is extremely helpful, it is not required to build a plugin. You can still build plugins without using the plugin object as you can see in the [Building a Panel Plugin](#) tutorial.

Info The Plugin object is only used for Panel Providers. Standalone Plugins do not use this object. All configuration for Standalone Plugins should be handled in the plugin's service provider.

Registering Assets

All [asset registration](#), including CSS, JS and Alpine Components, should be done through the plugin's service provider in the [packageBooted\(\)](#) method. This allows Filament to register the assets with the Asset Manager and load them when needed.

Creating a Plugin

While you can certainly build plugins from scratch, we recommend using the [Filament Plugin Skeleton](#) to quickly get started. This skeleton includes all the necessary boilerplate to get you up and running quickly.

Usage

To use the skeleton, simply go to the GitHub repo and click the "Use this template" button. This will create a new repo in your account with the skeleton code. After that, you can clone the repo to your machine. Once you have the code on your machine, navigate to the root of the project and run the following command:

```
php ./configure.php
```

This will ask you a series of questions to configure the plugin. Once you've answered all the questions, the script will stub out a new plugin for you, and you can begin to build your amazing new extension for Filament.

Upgrading existing plugins

Since every plugin varies greatly in its scope of use and functionality, there is no one size fits all approaches to upgrading existing plugins. However, one thing to note, that is consistent to all plugins is the deprecation of the [PluginServiceProvider](#).

In your plugin service provider, you will need to change it to extend the PackageServiceProvider instead. You will also need to add a static `$name` property to the service provider. This property is used to register the plugin with Filament. Here is an example of what your service provider might look like:

```
class MyPluginServiceProvider extends PackageServiceProvider
{
    public static string $name = 'my-plugin';

    public function configurePackage(Package $package): void
    {
        $package->name(static::$name);
    }
}
```

Helpful links

Please read this guide in its entirety before upgrading your plugin. It will help you understand the concepts and how to build your plugin.

1. [Filament Asset Management](#)
2. [Panel Plugin Development](#)
3. [Icon Management](#)
4. [Colors Management](#)
5. [Style Customization](#)

Build A Panel Plugin

Preface

Please read the docs on [panel plugin development](#) and the [getting started guide](#) before continuing.

Overview

In this walkthrough, we'll build a simple plugin that adds a new form field that can be used in forms. This also means it will be available to users in their panels.

You can find the final code for this plugin at <https://github.com/awcodes/clock-widget>.

Step 1: Create the plugin

First, we'll create the plugin using the steps outlined in the [getting started guide](#).

Step 2: Clean up

Next, we'll clean up the plugin to remove the boilerplate code we don't need. This will seem like a lot, but since this is a simple plugin, we can remove a lot of the boilerplate code.

Remove the following directories and files:

1. config
2. database
3. src/Commands
4. src/Facades
5. stubs

Since our plugin doesn't have any settings or additional methods needed for functionality, we can also remove the `ClockWidgetPlugin.php` file.

1. ClockWidgetPlugin.php

Since Filament v3 recommends that users style their plugins with a custom filament theme, we'll remove the files needed for using css in the plugin. This is optional, and you can still use css if you want, but it is not recommended.

1. resources/css
2. postcss.config.js
3. tailwind.config.js

Now we can clean up our `composer.json` file to remove unneeded options.

```

"autoload": {
    "psr-4": {
        // We can remove the database factories
        "Awcodes\\ClockWidget\\Database\\Factories\\": "database/factories/"
    }
},
"extra": {
    "laravel": {
        // We can remove the facade
        "aliases": {
            "ClockWidget": "Awcodes\\ClockWidget\\Facades\\ClockWidget"
        }
    }
},

```

The last step is to update the `package.json` file to remove unneeded options. Replace the contents of `package.json` with the following.

```
{
    "private": true,
    "type": "module",
    "scripts": {
        "dev": "node bin/build.js --dev",
        "build": "node bin/build.js"
    },
    "devDependencies": {
        "esbuild": "^0.17.19"
    }
}
```

Then we need to install our dependencies.

```
npm install
```

You may also remove the Testing directories and files, but we'll leave them in for now, although we won't be using them for this example, and we highly recommend that you write tests for your plugins.

Step 3: Setting up the provider

Now that we have our plugin cleaned up, we can start adding our code. The boilerplate in the `src/ClockWidgetServiceProvider.php` file has a lot going on so, let's delete everything and start from scratch.

In this example, we will be registering an [async Alpine component](#). Since these assets are only loaded on request, we can register them as normal in the `packageBooted()` method. If you are registering assets, like CSS or JS files, that get loaded on every page regardless of if they are used or not, you should register them in the `register()` method of the `Plugin` configuration object, using `$panel->assets()`. Otherwise, if you register them in the `packageBooted()` method, they will be loaded in every panel, regardless of whether or not the plugin has been registered for that panel.

We need to be able to register our Widget with the panel and load our Alpine component when the widget is used. To do this, we'll need to add the following to the `packageBooted()` method in our service provider. This will register our widget component with Livewire and our Alpine component with the Filament Asset Manager.

```

use Filament\Support\Assets\AlpineComponent;
use Filament\Support\Facades\FilamentAsset;
use Livewire\Livewire;
use Spatie\LaravelPackageTools\Package;
use Spatie\LaravelPackageTools\PackageServiceProvider;

class ClockWidgetServiceProvider extends PackageServiceProvider
{
    public static string $name = 'clock-widget';

    public function configurePackage(Package $package): void
    {
        $package->name(static::$name)
            ->hasViews()
            ->hasTranslations();
    }

    public function packageBooted(): void
    {
        Livewire::component('clock-widget', ClockWidget::class);

        // Asset Registration
        FilamentAsset::register(
            assets:[
                AlpineComponent::make('clock-widget', __DIR__ . '/../resources/dist/clock-
widget.js'),
            ],
            package: 'awcodes/clock-widget'
        );
    }
}

```

Step 4: Create the widget

Now we can create our widget. We'll first need to extend Filament's `Widget` class in our `ClockWidget.php` file and tell it where to find the view for the widget. Since we are using the `PackageServiceProvider` to register our views, we can use the `:::` syntax to tell Filament where to find the view.

```

use Filament\Widgets\Widget;

class ClockWidget extends Widget
{
    protected static string $view = 'clock-widget::widget';
}

```

Next, we'll need to create the view for our widget. Create a new file at `resources/views/widget.blade.php` and add the following code. We'll make use of Filament's blade components to save time on writing the html for the widget.

We are using `async` `Alpine` to load our `Alpine` component, so we'll need to add the `x-ignore` attribute to the `div` that will load our component. We'll also need to add the `ax-load` attribute to the `div` to tell `Alpine` to load our component. You can learn more about this in the [Core Concepts](#) section of the docs.

```
<x-filament-widgets::widget>
  <x-filament::section>
    <x-slot name="heading">
      {{ __('clock-widget::clock-widget.title') }}
    </x-slot>

    <div
      x-ignore
      ax-load
      ax-load-src="{!!
\Filament\Support\Facades\FilamentAsset::getAlpineComponentSrc('clock-widget', 'awcodes/clock-
widget') !!}"
      x-data="clockWidget()"
      class="text-center"
    >
      <p>{{ __('clock-widget::clock-widget.description') }}</p>
      <p class="text-xl" x-text="time"></p>
    </div>
  </x-filament::section>
</x-filament-widgets::widget>
```

Next, we need to write our Alpine component in `src/js/index.js`. And build our assets with `npm run build`.

```
export default function clockWidget() {
  return {
    time: new Date().toLocaleTimeString(),
    init() {
      setInterval(() => {
        this.time = new Date().toLocaleTimeString();
      }, 1000);
    }
  }
}
```

We should also add translations for the text in the widget so users can translate the widget into their language. We'll add the translations to `resources/lang/en/widget.php`.

```
return [
  'title' => 'Clock Widget',
  'description' => 'Your current time is:',
];
```

Step 5: Update your README

You'll want to update your `README.md` file to include instructions on how to install your plugin and any other information you want to share with users. Like how to use it in their projects. For example:

```
// Register the plugin and/or Widget in your Panel provider:  
  
use Awcodes\ClockWidget\ClockWidgetWidget;  
  
public function panel(Panel $panel): Panel  
{  
    return $panel  
        ->widgets([  
            ClockWidgetWidget::class,  
        ]);  
}
```

And, that's it, our users can now install our plugin and use it in their projects.

Build A Standalone Plugin

Preface

Please read the docs on [panel plugin development](#) and the [getting started guide](#) before continuing.

Overview

In this walkthrough, we'll build a simple plugin that adds a new form component that can be used in forms. This also means it will be available to users in their panels.

You can find the final code for this plugin at <https://github.com/awcodes/headings>.

Step 1: Create the plugin

First, we'll create the plugin using the steps outlined in the [getting started guide](#).

Step 2: Clean up

Next, we'll clean up the plugin to remove the boilerplate code we don't need. This will seem like a lot, but since this is a simple plugin, we can remove a lot of the boilerplate code.

Remove the following directories and files:

1. `bin`
2. `config`
3. `database`
4. `src/Commands`
5. `src/Facades`
6. `stubs`
7. `tailwind.config.js`

Now we can clean up our `composer.json` file to remove unneeded options.

```
"autoload": {
    "psr-4": {
        // We can remove the database factories
        "Awcodess\\Headings\\Database\\Factories\\": "database/factories/"
    }
},
"extra": {
    "laravel": {
        // We can remove the facade
        "aliases": {
            "Headings": "Awcodess\\Headings\\Facades\\ClockWidget"
        }
    }
},
```

Normally, Filament v3 recommends that users style their plugins with a custom filament theme, but for the sake of example let's provide our own stylesheet that can be loaded asynchronously using the new `x-load` features in Filament v3. So, let's update our `package.json` file to include cssnano, postcss, postcss-cli and postcss-nesting to build our stylesheet.

```
{
    "private": true,
    "scripts": {
        "build": "postcss resources/css/index.css -o resources/dist/headings.css"
    },
    "devDependencies": {
        "cssnano": "^6.0.1",
        "postcss": "^8.4.27",
        "postcss-cli": "^10.1.0",
        "postcss-nesting": "^12.0.0"
    }
}
```

Then we need to install our dependencies.

```
npm install
```

We will also need to update our `postcss.config.js` file to configure postcss.

```
module.exports = {
    plugins: [
        require('postcss-nesting')(),
        require('cssnano')({
            preset: 'default',
        }),
    ],
};
```

You may also remove the testing directories and files, but we'll leave them in for now, although we won't be using them for this example, and we highly recommend that you write tests for your plugins.

Step 3: Setting up the provider

Now that we have our plugin cleaned up, we can start adding our code. The boilerplate in the `src/HeadingsServiceProvider.php` file has a lot going on so, let's delete everything and start from scratch.

We need to be able to register our stylesheet with the Filament Asset Manager so that we can load it on demand in our blade view. To do this, we'll need to add the following to the `packageBooted` method in our service provider.

Note the `loadedOnRequest()` method. This is important, because it tells Filament to only load the stylesheet when it's needed.

```
namespace Awcodes\Headings;

use Filament\Support\Assets\Css;
use Filament\Support\Facades\FilamentAsset;
use Spatie\LaravelPackageTools\Package;
use Spatie\LaravelPackageTools\PackageServiceProvider;

class HeadingsServiceProvider extends PackageServiceProvider
{
    public static string $name = 'headings';

    public function configurePackage(Package $package): void
    {
        $package->name(static::$name)
            ->hasViews();
    }

    public function packageBooted(): void
    {
        FilamentAsset::register([
            Css::make('headings', __DIR__ . '/../resources/dist/headings.css')-
>loadedOnRequest(),
        ], 'awcodes/headings');
    }
}
```

Step 4: Creating our component

Next, we'll need to create our component. Create a new file at `src/Heading.php` and add the following code.

```

namespace Awcodes\Headings;

use Closure;
use Filament\Forms\Components\Component;
use Filament\Support\Colors\Color;
use Filament\Support\Concerns\HasColor;

class Heading extends Component
{
    use HasColor;

    protected string | int $level = 2;

    protected string | Closure $content = '';

    protected string $view = 'headings::heading';

    final public function __construct(string | int $level)
    {
        $this->level($level);
    }

    public static function make(string | int $level): static
    {
        return app(static::class, ['level' => $level]);
    }

    protected function setUp(): void
    {
        parent::setUp();

        $this->dehydrated(false);
    }

    public function content(string | Closure $content): static
    {
        $this->content = $content;

        return $this;
    }

    public function level(string | int $level): static
    {
        $this->level = $level;

        return $this;
    }

    public function getColor(): array
    {
        return $this->evaluate($this->color) ?? Color::Amber;
    }

    public function getContent(): string
    {
        return $this->evaluate($this->content);
    }
}

```

```

    }

    public function getLevel(): string
    {
        return is_int($this->level) ? 'h' . $this->level : $this->level;
    }
}

```

Step 5: Rendering our component

Next, we'll need to create the view for our component. Create a new file at `resources/views/heading.blade.php` and add the following code.

We are using x-load to asynchronously load stylesheet, so it's only loaded when necessary. You can learn more about this in the [Core Concepts](#) section of the docs.

```

@php
$level = $getLevel();
$color = $getColor();
@endphp

<{{ $level }}>
  x-data
  x-load-css="@js(\Filament\Support\Facades\FilamentAsset::getStyleHref('headings', package: 'awcodes/headings'))"
  {{
    $attributes
    ->class([
      'headings-component',
      match ($color) {
        'gray' => 'text-gray-600 dark:text-gray-400',
        default => 'text-custom-500',
      },
    ])
    ->style([
      \Filament\Support\get_color_css_variables($color, [500]) => $color !== 'gray',
    ])
  }}
>
  {{ $getContent() }}
</{{ $level }}>

```

Step 6: Adding some styles

Next, let's provide some custom styling for our field. We'll add the following to `resources/css/index.css`. And run `npm run build` to compile our css.

```
.headings-component {
  &:is(h1, h2, h3, h4, h5, h6) {
    font-weight: 700;
    letter-spacing: -.025em;
    line-height: 1.1;
  }

  &h1 {
    font-size: 2rem;
  }

  &h2 {
    font-size: 1.75rem;
  }

  &h3 {
    font-size: 1.5rem;
  }

  &h4 {
    font-size: 1.25rem;
  }

  &h5,
  &h6 {
    font-size: 1rem;
  }
}
```

Then we need to build our stylesheet.

```
npm run build
```

Step 7: Update your README

You'll want to update your `README.md` file to include instructions on how to install your plugin and any other information you want to share with users, like how to use it in their projects. For example:

```
use Awcodes\Headings\Heading;

Heading::make(2)
->content('Product Information')
->color(Color::Lime),
```

And, that's it, our users can now install our plugin and use it in their projects.

Blade Components

Overview

Overview

Filament packages consume a set of core Blade components that aim to provide a consistent and maintainable foundation for all interfaces. Some of these components are also available for use in your own applications and Filament plugins.

Available UI components

- [Avatar](#)
- [Badge](#)
- [Breadcrumbs](#)
- [Loading indicator](#)
- [Section](#)
- [Tabs](#)

UI components for actions

- [Button](#)
- [Dropdown](#)
- [Icon button](#)
- [Link](#)
- [Modal](#)

UI components for forms

- [Checkbox](#)
- [Fieldset](#)
- [Input](#)
- [Input wrapper](#)
- [Select](#)

UI components for tables

- [Pagination](#)

Avatar

Overview

The avatar component is used to render a circular or square image, often used to represent a user or entity as their "profile picture":

```
<x-filament::avatar  
    src="https://filamentphp.com/dan.jpg"  
    alt="Dan Harrin"  
/>
```

Setting the rounding of an avatar

Avatars are fully rounded by default, but you may make them square by setting the `circular` attribute to `false`:

```
<x-filament::avatar  
    src="https://filamentphp.com/dan.jpg"  
    alt="Dan Harrin"  
    :circular="false"  
/>
```

Setting the size of an avatar

By default, the avatar will be "medium" size. You can set the size to either `sm`, `md`, or `lg` using the `size` attribute:

```
<x-filament::avatar  
    src="https://filamentphp.com/dan.jpg"  
    alt="Dan Harrin"  
    size="lg"  
/>
```

You can also pass your own custom size classes into the `size` attribute:

```
<x-filament::avatar  
    src="https://filamentphp.com/dan.jpg"  
    alt="Dan Harrin"  
    size="w-12 h-12"  
/>
```

Badge

Overview

The badge component is used to render a small box with some text inside:

```
<x-filament::badge>
  New
</x-filament::badge>
```

Setting the size of a badge

By default, the size of a badge is "medium". You can make it "extra small" or "small" by using the `size` attribute:

```
<x-filament::badge size="xs">
  New
</x-filament::badge>

<x-filament::badge size="sm">
  New
</x-filament::badge>
```

Changing the color of the badge

By default, the color of a badge is "primary". You can change it to be `danger`, `gray`, `info`, `success` or `warning` by using the `color` attribute:

```
<x-filament::badge color="danger">
  New
</x-filament::badge>

<x-filament::badge color="gray">
  New
</x-filament::badge>

<x-filament::badge color="info">
  New
</x-filament::badge>

<x-filament::badge color="success">
  New
</x-filament::badge>

<x-filament::badge color="warning">
  New
</x-filament::badge>
```

Adding an icon to a badge

You can add an `icon` to a badge by using the `icon` attribute:

```
<x-filament::badge icon="heroicon-m-sparkles">  
  New  
</x-filament::badge>
```

You can also change the icon's position to be after the text instead of before it, using the `icon-position` attribute:

```
<x-filament::badge  
  icon="heroicon-m-sparkles"  
  icon-position="after"  
>  
  New  
</x-filament::badge>
```

Breadcrumbs

Overview

The breadcrumbs component is used to render a simple, linear navigation that informs the user of their current location within the application:

```
<x-filament::breadcrumbs :breadcrumbs="[
    '/' => 'Home',
    '/dashboard' => 'Dashboard',
    '/dashboard/users' => 'Users',
    '/dashboard/users/create' => 'Create User',
]" />
```

The keys of the array are URLs that the user is able to click on to navigate, and the values are the text that will be displayed for each link.

Button

Overview

The button component is used to render a clickable button that can perform an action:

```
<x-filament::button wire:click="openNewUserModal">
    New user
</x-filament::button>
```

Using a button as an anchor link

By default, a button's underlying HTML tag is `<button>`. You can change it to be an `<a>` tag by using the `tag` attribute:

```
<x-filament::button
    href="https://filamentphp.com"
    tag="a"
>
    Filament
</x-filament::button>
```

Setting the size of a button

By default, the size of a button is "medium". You can make it "extra small", "small", "large" or "extra large" by using the `size` attribute:

```
<x-filament::button size="xs">
    New user
</x-filament::button>

<x-filament::button size="sm">
    New user
</x-filament::button>

<x-filament::button size="lg">
    New user
</x-filament::button>

<x-filament::button size="xl">
    New user
</x-filament::button>
```

Changing the color of a button

By default, the color of a button is "primary". You can change it to be `danger`, `gray`, `info`, `success` or `warning` by using the `color` attribute:

```
<x-filament::button color="danger">
    New user
</x-filament::button>

<x-filament::button color="gray">
    New user
</x-filament::button>

<x-filament::button color="info">
    New user
</x-filament::button>

<x-filament::button color="success">
    New user
</x-filament::button>

<x-filament::button color="warning">
    New user
</x-filament::button>
```

Adding an icon to a button

You can add an `icon` to a button by using the `icon` attribute:

```
<x-filament::button icon="heroicon-m-sparkles">
    New user
</x-filament::button>
```

You can also change the icon's position to be after the text instead of before it, using the `icon-position` attribute:

```
<x-filament::button
    icon="heroicon-m-sparkles"
    icon-position="after"
>
    New user
</x-filament::button>
```

Making a button outlined

You can make a button use an "outlined" design using the `outlined` attribute:

```
<x-filament::button outlined>
    New user
</x-filament::button>
```

Adding a tooltip to a button

You can add a tooltip to a button by using the `tooltip` attribute:

```
<x-filament::button tooltip="Register a user">
  New user
</x-filament::button>
```

Adding a badge to a button

You can render a badge on top of a button by using the `badge` slot:

```
<x-filament::button>
  Mark notifications as read

  <x-slot name="badge">
    3
  </x-slot>
</x-filament::button>
```

You can change the color of the badge using the `badge-color` attribute:

```
<x-filament::button badge-color="danger">
  Mark notifications as read

  <x-slot name="badge">
    3
  </x-slot>
</x-filament::button>
```

Checkbox

Overview

You can use the checkbox component to render a checkbox input that can be used to toggle a boolean value:

```
<label>
    <x-filament::input.checkbox wire:model="isAdmin" />

    <span>
        Is Admin
    </span>
</label>
```

Triggering the error state of the checkbox

The checkbox has special styling that you can use if it is invalid. To trigger this styling, you can use either Blade or Alpine.js.

To trigger the error state using Blade, you can pass the `:valid` attribute to the component, which contains either true or false based on if the checkbox is valid or not:

```
<x-filament::input.checkbox
    wire:model="isAdmin"
    :valid="! $errors->has('isAdmin')"
/>
```

Alternatively, you can use an Alpine.js expression to trigger the error state, based on if it evaluates to `true` or `false`:

```
<div x-data="{ errors: ['isAdmin'] }">
    <x-filament::input.checkbox
        x-model="isAdmin"
        alpine-valid="! errors.includes('isAdmin')"
    />
</div>
```

Dropdown

Overview

The dropdown component allows you to render a dropdown menu with a button that triggers it:

```
<x-filament::dropdown>
  <x-slot name="trigger">
    <x-filament::button>
      More actions
    </x-filament::button>
  </x-slot>

  <x-filament::dropdown.list>
    <x-filament::dropdown.list.item wire:click="openViewModal">
      View
    </x-filament::dropdown.list.item>

    <x-filament::dropdown.list.item wire:click="openEditModal">
      Edit
    </x-filament::dropdown.list.item>

    <x-filament::dropdown.list.item wire:click="openDeleteModal">
      Delete
    </x-filament::dropdown.list.item>
  </x-filament::dropdown.list>
</x-filament::dropdown>
```

Using a dropdown item as an anchor link

By default, a dropdown item's underlying HTML tag is `<button>`. You can change it to be an `<a>` tag by using the `tag` attribute:

```
<x-filament::dropdown.list.item
  href="https://filamentphp.com"
  tag="a">
>
  Filament
</x-filament::dropdown.list.item>
```

Changing the color of a dropdown item

By default, the color of a dropdown item is "gray". You can change it to be `danger`, `info`, `primary`, `success` or `warning` by using the `color` attribute:

```
<x-filament::dropdown.list.item color="danger">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item color="info">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item color="primary">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item color="success">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item color="warning">
    Edit
</x-filament::dropdown.list.item>
```

Adding an icon to a dropdown item

You can add an icon to a dropdown item by using the `icon` attribute:

```
<x-filament::dropdown.list.item icon="heroicon-m-pencil">
    Edit
</x-filament::dropdown.list.item>
```

Changing the icon color of a dropdown item

By default, the icon color uses the same color as the item itself. You can override it to be `danger`, `info`, `primary`, `success` or `warning` by using the `icon-color` attribute:

```
<x-filament::dropdown.list.item icon="heroicon-m-pencil" icon-color="danger">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item icon="heroicon-m-pencil" icon-color="info">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item icon="heroicon-m-pencil" icon-color="primary">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item icon="heroicon-m-pencil" icon-color="success">
    Edit
</x-filament::dropdown.list.item>

<x-filament::dropdown.list.item icon="heroicon-m-pencil" icon-color="warning">
    Edit
</x-filament::dropdown.list.item>
```

Adding an image to a dropdown item

You can add a circular image to a dropdown item by using the `image` attribute:

```
<x-filament::dropdown.list.item image="https://filamentphp.com/dan.jpg">
    Dan Harrin
</x-filament::dropdown.list.item>
```

Adding a badge to a dropdown item

You can render a `badge` on top of a dropdown item by using the `badge` slot:

```
<x-filament::dropdown.list.item>
    Mark notifications as read

    <x-slot name="badge">
        3
    </x-slot>
</x-filament::dropdown.list.item>
```

You can change the color of the badge using the `badge-color` attribute:

```
<x-filament::dropdown.list.item badge-color="danger">
    Mark notifications as read

    <x-slot name="badge">
        3
    </x-slot>
</x-filament::dropdown.list.item>
```

Setting the placement of a dropdown

The dropdown may be positioned relative to the trigger button by using the `placement` attribute:

```
<x-filament::dropdown placement="top-start">
    {{-- Dropdown items --}}
</x-filament::dropdown>
```

Setting the width of a dropdown

The dropdown may be set to a width by using the `width` attribute. Options correspond to [Tailwind's max-width scale](#).

The options are `xs`, `sm`, `md`, `lg`, `xl`, `2xl`, `3xl`, `4xl`, `5xl`, `6xl` and `7xl`:

```
<x-filament::dropdown width="xs">
    {{-- Dropdown items --}}
</x-filament::dropdown>
```

Controlling the maximum height of a dropdown

The dropdown content can have a maximum height using the `max-height` attribute, so that it scrolls. You can pass a [CSS length](#):

```
<x-filament::dropdown max-height="400px">
  {{-- Dropdown items --}}
</x-filament::dropdown>
```

Fieldset

Overview

You can use a fieldset to group multiple form fields together, optionally with a label:

```
<x-filament::fieldset>
  <x-slot name="label">
    Address
  </x-slot>

  {{-- Form fields --}}
</x-filament::fieldset>
```

Icon Button

Overview

The button component is used to render a clickable button that can perform an action:

```
<x-filament::icon-button
    icon="heroicon-m-plus"
    wire:click="openNewUserModal"
    label="New label"
/>
```

Using an icon button as an anchor link

By default, an icon button's underlying HTML tag is `<button>`. You can change it to be an `<a>` tag by using the `tag` attribute:

```
<x-filament::icon-button
    icon="heroicon-m-arrow-top-right-on-square"
    href="https://filamentphp.com"
    tag="a"
    label="Filament"
/>
```

Setting the size of an icon button

By default, the size of an icon button is "medium". You can make it "extra small", "small", "large" or "extra large" by using the `size` attribute:

```
<x-filament::icon-button
  icon="heroicon-m-plus"
  size="xs"
  label="New label"
/>

<x-filament::icon-button
  icon="heroicon-m-plus"
  size="sm"
  label="New label"
/>

<x-filament::icon-button
  icon="heroicon-s-plus"
  size="lg"
  label="New label"
/>

<x-filament::icon-button
  icon="heroicon-s-plus"
  size="xl"
  label="New label"
/>
```

Changing the color of an icon button

By default, the color of an icon button is "primary". You can change it to be `danger`, `gray`, `info`, `success` or `warning` by using the `color` attribute:

```
<x-filament::icon-button
    icon="heroicon-m-plus"
    color="danger"
    label="New label"
/>

<x-filament::icon-button
    icon="heroicon-m-plus"
    color="gray"
    label="New label"
/>

<x-filament::icon-button
    icon="heroicon-m-plus"
    color="info"
    label="New label"
/>

<x-filament::icon-button
    icon="heroicon-m-plus"
    color="success"
    label="New label"
/>

<x-filament::icon-button
    icon="heroicon-m-plus"
    color="warning"
    label="New label"
/>
```

Adding a tooltip to an icon button

You can add a tooltip to an icon button by using the `tooltip` attribute:

```
<x-filament::icon-button
    icon="heroicon-m-plus"
    tooltip="Register a user"
    label="New label"
/>
```

Adding a badge to an icon button

You can render a `badge` on top of an icon button by using the `badge` slot:

```
<x-filament::icon-button
    icon="heroicon-m-x-mark"
    label="Mark notifications as read"
>
<x-slot name="badge">
    3
</x-slot>
</x-filament::icon-button>
```

You can change the color of the badge using the `badge-color` attribute:

```
<x-filament::icon-button
  icon="heroicon-m-x-mark"
  label="Mark notifications as read"
  badge-color="danger"
>
  <x-slot name="badge">
    3
  </x-slot>
</x-filament::icon-button>
```

Input Wrapper

Overview

The input wrapper component should be used as a wrapper around the `input` or `select` components. It provides a border and other elements such as a prefix or suffix.

```
<x-filament::input.wrapper>
  <x-filament::input
    type="text"
    wire:model="name"
  />
</x-filament::input.wrapper>

<x-filament::input.wrapper>
  <x-filament::input.select wire:model="status">
    <option value="draft">Draft</option>
    <option value="reviewing">Reviewing</option>
    <option value="published">Published</option>
  </x-filament::input.select>
</x-filament::input.wrapper>
```

Triggering the error state of the input

The component has special styling that you can use if it is invalid. To trigger this styling, you can use either Blade or Alpine.js.

To trigger the error state using Blade, you can pass the `valid` attribute to the component, which contains either true or false based on if the input is valid or not:

```
<x-filament::input.wrapper :valid="! $errors->has('name')">
  <x-filament::input
    type="text"
    wire:model="name"
  />
</x-filament::input.wrapper>
```

Alternatively, you can use an Alpine.js expression to trigger the error state, based on if it evaluates to `true` or `false`:

```
<div x-data="{ errors: ['name'] }">
  <x-filament::input.wrapper alpine-valid="! errors.includes('name')">
    <x-filament::input
      type="text"
      wire:model="name"
    />
  </x-filament::input.wrapper>
</div>
```

Disabling the input

To disable the input, you must also pass the `disabled` attribute to the wrapper component:

```
<x-filament::input.wrapper disabled>
  <x-filament::input
    type="text"
    wire:model="name"
    disabled
  />
</x-filament::input.wrapper>
```

Adding affix text aside the input

You may place text before and after the input using the `prefix` and `suffix` slots:

```
<x-filament::input.wrapper>
  <x-slot name="prefix">
    https://
  </x-slot>

  <x-filament::input
    type="text"
    wire:model="domain"
  />

  <x-slot name="suffix">
    .com
  </x-slot>
</x-filament::input.wrapper>
```

Using icons as affixes

You may place an `icon` before and after the input using the `prefix-icon` and `suffix-icon` attributes:

```
<x-filament::input.wrapper suffix-icon="heroicon-m-globe-alt">
  <x-filament::input
    type="url"
    wire:model="domain"
  />
</x-filament::input.wrapper>
```

Setting the affix icon's color

Affix icons are gray by default, but you may set a different color using the `prefix-icon-color` and `affix-icon-color` attributes:

```
<x-filament::input.wrapper
  suffix-icon="heroicon-m-check-circle"
  suffix-icon-color="success"
>
<x-filament::input
  type="url"
  wire:model="domain"
/>
</x-filament::input.wrapper>
```

Input

Overview

The input component is a wrapper around the native `<input>` element. It provides a simple interface for entering a single line of text.

```
<x-filament::input.wrapper>
  <x-filament::input
    type="text"
    wire:model="name"
  />
</x-filament::input.wrapper>
```

To use the input component, you must wrap it in an "input wrapper" component, which provides a border and other elements such as a prefix or suffix. You can learn more about customizing the input wrapper component [here](#).

Link

Overview

The link component is used to render a clickable link that can perform an action:

```
<x-filament::link :href="route('users.create')">
    New user
</x-filament::link>
```

Using a link as a button

By default, a link's underlying HTML tag is `<a>`. You can change it to be a `<button>` tag by using the `tag` attribute:

```
<x-filament::link
    wire:click="openNewUserModal"
    tag="button"
>
    New user
</x-filament::link>
```

Setting the size of a link

By default, the size of a link is "medium". You can make it "small", "large", "extra large" or "extra extra large" by using the `size` attribute:

```
<x-filament::link size="sm">
    New user
</x-filament::link>

<x-filament::link size="lg">
    New user
</x-filament::link>

<x-filament::link size="xl">
    New user
</x-filament::link>

<x-filament::link size="2xl">
    New user
</x-filament::link>
```

Setting the font weight of a link

By default, the font weight of links is `semibold`. You can make it `thin`, `extralight`, `light`, `normal`, `medium`, `bold`, `extrabold` or `black` by using the `weight` attribute:

```
<x-filament::link weight="thin">
    New user
</x-filament::link>

<x-filament::link weight="extralight">
    New user
</x-filament::link>

<x-filament::link weight="light">
    New user
</x-filament::link>

<x-filament::link weight="normal">
    New user
</x-filament::link>

<x-filament::link weight="medium">
    New user
</x-filament::link>

<x-filament::link weight="semibold">
    New user
</x-filament::link>

<x-filament::link weight="bold">
    New user
</x-filament::link>

<x-filament::link weight="black">
    New user
</x-filament::link>
```

Alternatively, you can pass in a custom CSS class to define the weight:

```
<x-filament::link weight="md:font-[650]">
    New user
</x-filament::link>
```

Changing the color of a link

By default, the color of a link is "primary". You can change it to be `danger`, `gray`, `info`, `success` or `warning` by using the `color` attribute:

```
<x-filament::link color="danger">
    New user
</x-filament::link>

<x-filament::link color="gray">
    New user
</x-filament::link>

<x-filament::link color="info">
    New user
</x-filament::link>

<x-filament::link color="success">
    New user
</x-filament::link>

<x-filament::link color="warning">
    New user
</x-filament::link>
```

Adding an icon to a link

You can add an `icon` to a link by using the `icon` attribute:

```
<x-filament::link icon="heroicon-m-sparkles">
    New user
</x-filament::link>
```

You can also change the icon's position to be after the text instead of before it, using the `icon-position` attribute:

```
<x-filament::link
    icon="heroicon-m-sparkles"
    icon-position="after"
>
    New user
</x-filament::link>
```

Adding a tooltip to a link

You can add a tooltip to a link by using the `tooltip` attribute:

```
<x-filament::link tooltip="Register a user">
    New user
</x-filament::link>
```

Adding a badge to a link

You can render a `badge` on top of a link by using the `badge` slot:

```
<x-filament::link>
  Mark notifications as read

  <x-slot name="badge">
    3
  </x-slot>
</x-filament::link>
```

You can change the color of the badge using the `badge-color` attribute:

```
<x-filament::link badge-color="danger">
  Mark notifications as read

  <x-slot name="badge">
    3
  </x-slot>
</x-filament::link>
```

Loading Indicator

Overview

The loading indicator is an animated SVG that can be used to indicate that something is in progress:

```
<x-filament::loading-indicator class="h-5 w-5" />
```

Modal

Overview

The modal component is able to open a dialog window or slide-over with any content:

```
<x-filament::modal>
  <x-slot name="trigger">
    <x-filament::button>
      Open modal
    </x-filament::button>
  </x-slot>

  {{-- Modal content --}}
</x-filament::modal>
```

Controlling a modal from JavaScript

You can use the `trigger` slot to render a button that opens the modal. However, this is not required. You have complete control over when the modal opens and closes through JavaScript. First, give the modal an ID so that you can reference it:

```
<x-filament::modal id="edit-user">
  {{-- Modal content --}}
</x-filament::modal>
```

Now, you can dispatch an `open-modal` or `close-modal` browser event, passing the modal's ID, which will open or close the modal. For example, from a Livewire component:

```
$this->dispatch('open-modal', id: 'edit-user');
```

Or from Alpine.js:

```
$dispatch('open-modal', { id: 'edit-user' })
```

Adding a heading to a modal

You can add a heading to a modal by using the `heading` slot:

```
<x-filament::modal>
  <x-slot name="heading">
    Modal heading
  </x-slot>

  {{-- Modal content --}}
</x-filament::modal>
```

Adding a description to a modal

You can add a description, below the heading, to a modal by using the `description` slot:

```
<x-filament::modal>
  <x-slot name="heading">
    Modal heading
  </x-slot>

  <x-slot name="description">
    Modal description
  </x-slot>

  {{-- Modal content --}}
</x-filament::modal>
```

Adding an icon to a modal

You can add an `icon` to a modal by using the `icon` attribute:

```
<x-filament::modal icon="heroicon-o-information-circle">
  <x-slot name="heading">
    Modal heading
  </x-slot>

  {{-- Modal content --}}
</x-filament::modal>
```

By default, the color of an icon is "primary". You can change it to be `danger`, `gray`, `info`, `success` or `warning` by using the `icon-color` attribute:

```
<x-filament::modal
  icon="heroicon-o-exclamation-triangle"
  icon-color="danger">
  <x-slot name="heading">
    Modal heading
  </x-slot>

  {{-- Modal content --}}
</x-filament::modal>
```

Adding a footer to a modal

You can add a footer to a modal by using the `footer` slot:

```
<x-filament::modal>
  {{-- Modal content --}}

  <x-slot name="footer">
    {{-- Modal footer content --}}
  </x-slot>
</x-filament::modal>
```

Alternatively, you can add actions into the footer by using the `footerActions` slot:

```
<x-filament::modal>
  {{{-- Modal content --}}}

  <x-slot name="footerActions">
    {{{-- Modal footer actions --}}}
  </x-slot>
</x-filament::modal>
```

Changing the modal's alignment

By default, modal content will be aligned to the start, or centered if the modal is `xs` or `sm` in `width`. If you wish to change the alignment of content in a modal, you can use the `alignment` attribute and pass it `start` or `center`:

```
<x-filament::modal alignment="center">
  {{{-- Modal content --}}
</x-filament::modal>
```

Using a slide-over instead of a modal

You can open a "slide-over" dialog instead of a modal by using the `slide-over` attribute:

```
<x-filament::modal slide-over>
  {{{-- Slide-over content --}}
</x-filament::modal>
```

Making the modal header sticky

The header of a modal scrolls out of view with the modal content when it overflows the modal size. However, slide-overs have a sticky modal that's always visible. You may control this behavior using the `sticky-header` attribute:

```
<x-filament::modal sticky-header>
  <x-slot name="heading">
    Modal heading
  </x-slot>

  {{{-- Modal content --}}
</x-filament::modal>
```

Making the modal footer sticky

The footer of a modal is rendered inline after the content by default. Slide-overs, however, have a sticky footer that always shows when scrolling the content. You may enable this for a modal too using the `sticky-footer` attribute:

```
<x-filament::modal sticky-footer>
  {{{-- Modal content --}}

  <x-slot name="footer">
    {{{-- Modal footer content --}}
  </x-slot>
</x-filament::modal>
```

Changing the modal width

You can change the width of the modal by using the `width` attribute. Options correspond to [Tailwind's max-width scale](#). The options are `xs`, `sm`, `md`, `lg`, `xl`, `2xl`, `3xl`, `4xl`, `5xl`, `6xl`, `7xl`, and `screen`:

```
<x-filament::modal width="5xl">
  {{-- Modal content --}}
</x-filament::modal>
```

Closing the modal by clicking away

By default, when you click away from a modal, it will close itself. If you wish to disable this behavior for a specific action, you can use the `close-by-clicking-away` attribute:

```
<x-filament::modal :close-by-clicking-away="false">
  {{-- Modal content --}}
</x-filament::modal>
```

Closing the modal by escaping

By default, when you press escape on a modal, it will close itself. If you wish to disable this behavior for a specific action, you can use the `close-by-escaping` attribute:

```
<x-filament::modal :close-by-escaping="false">
  {{-- Modal content --}}
</x-filament::modal>
```

Hiding the modal close button

By default, modals have a close button in the top right corner. You can remove the close button from the modal by using the `close-button` attribute:

```
<x-filament::modal :close-button="false">
  {{-- Modal content --}}
</x-filament::modal>
```

Preventing the modal from autofocusing

By default, modals will autofocus on the first focusable element when opened. If you wish to disable this behavior, you can use the `autofocus` attribute:

```
<x-filament::modal :autofocus="false">
  {{-- Modal content --}}
</x-filament::modal>
```

Pagination

Overview

The pagination component can be used in a Livewire Blade view only. It can render a list of paginated links:

```
use App\Models\User;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class ListUsers extends Component
{
    // ...

    public function render(): View
    {
        return view('livewire.list-users', [
            'users' => User::query()->paginate(10),
        ]);
    }
}
```

```
<x-filament::pagination :paginator="$users" />
```

Alternatively, you can use simple pagination or cursor pagination, which will just render a "previous" and "next" button:

```
use App\Models\User;

User::query()->simplePaginate(10)
User::query()->cursorPaginate(10)
```

Allowing the user to customize the number of items per page

You can allow the user to customize the number of items per page by passing an array of options to the `page-options` attribute. You must also define a Livewire property where the user's selection will be stored:

```

use App\Models\User;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class ListUsers extends Component
{
    public int | string $perPage = 10;

    // ...

    public function render(): View
    {
        return view('livewire.list-users', [
            'users' => User::query()->paginate($this->perPage),
        ]);
    }
}

```

```

<x-filament::pagination
    :paginator="$users"
    :page-options="[5, 10, 20, 50, 100, 'all']"
    :current-page-option-property="perPage"
/>

```

Displaying links to the first and the last page

Extreme links are the first and last page links. You can add them by passing the `extreme-links` attribute to the component:

```

<x-filament::pagination
    :paginator="$users"
    extreme-links
/>

```

Section

Overview

A section can be used to group content together, with an optional heading:

```
<x-filament::section>
  <x-slot name="heading">
    User details
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Adding a description to the section

You can add a description below the heading to the section by using the `description` slot:

```
<x-filament::section>
  <x-slot name="heading">
    User details
  </x-slot>

  <x-slot name="description">
    This is all the information we hold about the user.
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Adding an icon to the section header

You can add an `icon` to a section by using the `icon` attribute:

```
<x-filament::section icon="heroicon-o-user">
  <x-slot name="heading">
    User details
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Changing the color of the section icon

By default, the color of the section icon is "gray". You can change it to be `danger`, `info`, `primary`, `success` or `warning` by using the `icon-color` attribute:

```
<x-filament::section
    icon="heroicon-o-user"
    icon-color="info"
>
    <x-slot name="heading">
        User details
    </x-slot>

    {{-- Content --}}
</x-filament::section>
```

Changing the size of the section icon

By default, the size of the section icon is "large". You can change it to be "small" or "medium" by using the `icon-size` attribute:

```
<x-filament::section
    icon="heroicon-m-user"
    icon-size="sm"
>
    <x-slot name="heading">
        User details
    </x-slot>

    {{-- Content --}}
</x-filament::section>

<x-filament::section
    icon="heroicon-m-user"
    icon-size="md"
>
    <x-slot name="heading">
        User details
    </x-slot>

    {{-- Content --}}
</x-filament::section>
```

Adding content to the end of the header

You may render additional content at the end of the header, next to the heading and description, using the `headerEnd` slot:

```
<x-filament::section>
  <x-slot name="heading">
    User details
  </x-slot>

  <x-slot name="headerEnd">
    {{-- Input to select the user's ID --}}
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Making a section collapsible

You can make the content of a section collapsible by using the `collapsible` attribute:

```
<x-filament::section collapsible>
  <x-slot name="heading">
    User details
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Making a section collapsed by default

You can make a section collapsed by default by using the `collapsed` attribute:

```
<x-filament::section
  collapsible
  collapsed
>
  <x-slot name="heading">
    User details
  </x-slot>

  {{-- Content --}}
</x-filament::section>
```

Persisting collapsed sections

You can persist whether a section is collapsed in local storage using the `persist-collapsed` attribute, so it will remain collapsed when the user refreshes the page. You will also need a unique `id` attribute to identify the section from others, so that each section can persist its own collapse state:

```
<x-filament::section
    collapsible
    collapsed
    persist-collapsed
    id="user-details">
>
<x-slot name="heading">
    User details
</x-slot>

{{-- Content --}}
</x-filament::section>
```

Adding the section header aside the content instead of above it

You can change the position of the section header to be aside the content instead of above it by using the `aside` attribute:

```
<x-filament::section aside>
<x-slot name="heading">
    User details
</x-slot>

{{-- Content --}}
</x-filament::section>
```

Positioning the content before the header

You can change the position of the content to be before the header instead of after it by using the `content-before` attribute:

```
<x-filament::section
    aside
    content-before
>
<x-slot name="heading">
    User details
</x-slot>

{{-- Content --}}
</x-filament::section>
```

Select

Overview

The select component is a wrapper around the native `<select>` element. It provides a simple interface for selecting a single value from a list of options:

```
<x-filament::input.wrapper>
  <x-filament::input.select wire:model="status">
    <option value="draft">Draft</option>
    <option value="reviewing">Reviewing</option>
    <option value="published">Published</option>
  </x-filament::input.select>
</x-filament::input.wrapper>
```

To use the select component, you must wrap it in an "input wrapper" component, which provides a border and other elements such as a prefix or suffix. You can learn more about customizing the input wrapper component [here](#).

Tabs

Overview

The tabs component allows you to render a set of tabs, which can be used to toggle between multiple sections of content:

```
<x-filament::tabs label="Content tabs">
  <x-filament::tabs.item>
    Tab 1
  </x-filament::tabs.item>

  <x-filament::tabs.item>
    Tab 2
  </x-filament::tabs.item>

  <x-filament::tabs.item>
    Tab 3
  </x-filament::tabs.item>
</x-filament::tabs>
```

Triggering the active state of the tab

By default, tabs do not appear "active". To make a tab appear active, you can use the `active` attribute:

```
<x-filament::tabs>
  <x-filament::tabs.item active>
    Tab 1
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

You can also use the `active` attribute to make a tab appear active conditionally:

```
<x-filament::tabs>
  <x-filament::tabs.item
    :active="$activeTab === 'tab1'"
    wire:click="$set('activeTab', 'tab1')"
  >
    Tab 1
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Or you can use the `alpine-active` attribute to make a tab appear active conditionally using Alpine.js:

```
<x-filament::tabs x-data="{ activeTab: 'tab1' }">
  <x-filament::tabs.item
    alpine-active="activeTab === 'tab1'"
    x-on:click="activeTab = 'tab1'"
  >
    Tab 1
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Setting a tab icon

Tabs may have an icon, which you can set using the `icon` attribute:

```
<x-filament::tabs>
  <x-filament::tabs.item icon="heroicon-m-bell">
    Notifications
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Setting the tab icon position

The icon of the tab may be positioned before or after the label using the `icon-position` attribute:

```
<x-filament::tabs>
  <x-filament::tabs.item
    icon="heroicon-m-bell"
    icon-position="after"
  >
    Notifications
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Setting a tab badge

Tabs may have a badge, which you can set using the `badge` slot:

```
<x-filament::tabs>
  <x-filament::tabs.item>
    Notifications

    <x-slot name="badge">
      5
    </x-slot>
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Using a tab as an anchor link

By default, a tab's underlying HTML tag is `<button>`. You can change it to be an `<a>` tag by using the `tag` attribute:

```
<x-filament::tabs>
  <x-filament::tabs.item
    :href="route('notifications')"
    tag="a"
  >
    Notifications
  </x-filament::tabs.item>

  {{-- Other tabs --}}
</x-filament::tabs>
```

Stubs

Publishing the stubs

If you would like to customize the files that are generated by Filament, you can do so by publishing the "stubs" to your application. These are template files that you can modify to your own preferences.

To publish the stubs to the `stubs/filament` directory, run the following command:

```
php artisan vendor:publish --tag=filament-stubs
```

Support

We offer a variety of support options, mostly free of charge. If you need help, the community is here for you.

Discord

We are fortunate to have a growing community of Filament users that help each other out on our [Discord server](#). Join now, its free! We also have many dedicated channels in different languages. Currently, we have channels for the following languages:

- [#ar](#) - Arabic 
- [#de](#) - German 
- [#es](#) - Spanish 
- [#fa](#) - Farsi 
- [#fr](#) - French 
- [#id](#) - Indonesian 
- [#it](#) - Italian 
- [#nl](#) - Dutch 
- [#pt-br](#) - Portuguese (Brazil) 
- [#tr](#) - Turkish 
- [#ko](#) - Korean 

If you are missing a channel for your language, please let us know and we will create one for you.

GitHub

You can also reach out to us on our [GitHub community forum](#). Where our community members and maintainers are happy to help you out.

If you find a bug, you can open an [issue](#), and even donate to the bug fix by using the link automatically added to the bottom of every new issue description.

If you have a feature request, you can create a discussion on GitHub [following these instructions](#). If you are not planning to contribute the feature yourself but the core team adds it to the roadmap, an issue will be created which you are able to fast-track its development by donating money to it using the link added to the bottom of the issue description.

One-on-one private support & consulting (paid)

If you're looking for dedicated help with your Filament project, we're here for you. Whether you're a solo developer or running a large company, we provide support and development services that fit your needs. More information can be found on our [consulting page](#).

Laracasts

[Laracasts](#) has a dedicated [Filament help section](#) where you can ask questions and get help from their community.

Additionally on Laracasts you can find two excellent courses about Filament:

- [Rapid Laravel Apps With Filament](#)
- [Build Advanced Components for Filament](#) by one of the creators of Filament.

An active subscription may be required to access these parts of these courses.

Google

Since we make use of [AnswerOverflow](#) on our [Discord](#) server, you are often one Google search away from finding an answer to your question or at least a hint on how to solve your problem. You may also find results from [GitHub](#), the [Laracasts forum](#), or even Stack Overflow.

Helping others

We would like to encourage you to join any of the above platforms and help yourself and our community out. Additionally, we would like to encourage you to [contribute](#) to Filament itself. We are always looking for new contributors to help us improve Filament.