

filament

Filament 2.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

- **Admin Panel**

- [Installation](#)
- **Resources**
 - [Getting Started](#)
 - [Listing Records](#)
 - [Creating Records](#)
 - [Editing Records](#)
 - [Viewing Records](#)
 - [Deleting Records](#)
 - [Relation Managers](#)
 - [Global Search](#)
 - [Widgets](#)
 - [Custom Pages](#)
 - [Security](#)
- **Pages**
 - [Getting Started](#)
 - [Actions](#)
 - [Widgets](#)
- **Dashboard**
 - [Getting Started](#)
 - [Stats](#)
 - [Charts](#)
 - [Tables](#)
- [Navigation](#)
- [Users](#)
- [Notifications](#)
- [Appearance](#)
- [Plugins](#)
- [Testing](#)
- [Upgrade Guide](#)

- **Form Builder**

- [Installation](#)
- [Getting Started](#)
- [Fields](#)
- [Layout](#)
- [Validation](#)
- [Advanced](#)
- [Testing](#)

- **Table Builder**

- [Installation](#)
- [Getting Started](#)
- **Columns**
 - [Getting Started](#)

- [Text](#)
- [Icon](#)
- [Image](#)
- [Badge](#)
- [Tags](#)
- [Color](#)
- [Select](#)
- [Toggle](#)
- [Text Input](#)
- [Checkbox](#)
- [Custom](#)
- [Relationships](#)
- [Filters](#)
- [Actions](#)
- [Layout](#)
- [Testing](#)

- **Notifications**

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

- **Spatie Laravel Translatable Plugin**

- [Installation](#)
- [Getting Started](#)

- **Spatie Laravel Tags Plugin**

- [Installation](#)
- [Form Components](#)
- [Table Columns](#)

- **Spatie Laravel Settings Plugin**

- [Installation](#)
- [Getting Started](#)

- **Spatie Laravel Media Library Plugin**

- [Installation](#)
- [Form Components](#)
- [Table Columns](#)

Chapter 1

Admin Panel

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

This package is compatible with other Filament v2.x products. The [form builder](#), [table builder](#) and [notifications](#) come pre-installed with the package, and no other installation steps are required to use them within the admin panel.

Installation

To get started with the admin panel, you can install it using the command:

```
composer require filament/filament:^2.0"
```

Each time you upgrade Filament, you need to run the `filament:upgrade` command. We recommend adding this to your `composer.json`'s `post-update-cmd`:

```
"post-update-cmd": [
    // ...
    "@php artisan filament:upgrade"
],
```

If you don't have one, you may create a new user account using:

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

Visit your admin panel at `/admin` to sign in, and you're now ready to start [building resources!](#)

The screenshot shows the Filament 2.x dashboard. On the left sidebar, there are sections for SHOP (Products, Customers, Orders, Categories, Brands, Discounts, Reviews), BLOG (Posts, Categories, Authors), and a user profile for 'Demo User'. The main dashboard area has a 'Welcome, Demo User' message and a 'Sign out' link. It features three cards: 'Revenue' (\$192.1k, 32k increase), 'New customers' (1340, 3% decrease), and 'New orders' (3543, 7% increase). Below these are two line charts: 'Orders per month' (orders from Jan to Dec) and 'Total customers' (customers from Jan to Nov).

Deploying to production

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 admin panel.

Please see the [Users page](#).

If you don't complete these steps, there will be a 403 error when you try to access the admin panel in production.

Publishing configuration

If you wish, you may publish the configuration of the package using:

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

Publishing translations

If you wish to translate the package, you may publish the language files using:

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

Since this package depends on other Filament packages, you may wish to translate those as well:

```
php artisan vendor:publish --tag=forms-translations
php artisan vendor:publish --tag=tables-translations
php artisan vendor:publish --tag=filament-support-translations
```

Upgrading

To upgrade the package to the latest version, you must run:

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

We recommend adding the `filament:upgrade` command to your `composer.json`'s `post-update-cmd` to run it automatically:

```
"post-update-cmd": [  
    // ...  
    "@php artisan filament:upgrade"  
,
```

Resources

Getting Started

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 admin panel 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.

The `doctrine/dbal` package is required to use this functionality:

```
composer require doctrine/dbal --dev
```

When creating your resource, you may now use `--generate`:

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

Note: If your table contains ENUM columns, `doctrine/dbal` is unable to scan your table and will crash. Hence Filament is unable to generate the schema for your resource if it contains an ENUM column. Read more about this issue [here](#).

Handling soft deletes

By default, you will not be able to interact with deleted records in the admin panel. 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
```

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.

Forms

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

```

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

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

```

Fields

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

We have many fields available for your forms, including:

- [Text input](#)
- [Select](#)
- [Multi-select](#)
- [Checkbox](#)
- [Date-time picker](#)
- [File upload](#)
- [Rich editor](#)
- [Markdown editor](#)
- [Repeater](#)

To view a full list of available form [fields](#), see the [Form Builder documentation](#).

You may also build your own completely [custom form fields](#).

Layout

Form layouts are completely customizable. We have many layout components available, which can be used in any combination:

- [Grid](#)
- [Card](#)
- [Tabs](#)

To view a full list of available [layout components](#), see the [Form Builder documentation](#).

You may also build your own completely [custom layout components](#).

Hiding components contextually

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'),
```

Tables

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

```
use Filament\Resources\Table;
use Filament\Tables;
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>DeleteBulkAction::make(),
        ]);
}
```

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

Relations

Filament has many utilities available for managing resource relationships. Which solution you choose to use depends on your use case:

BelongsTo**Select field**

Filament includes the ability to automatically load options from a **BelongsTo** relationship:

```
use Filament\Forms\Components\Select;

Select::make('author_id')
->relationship('author', 'name')
```

More information is available in the [Form docs](#).

Layout component

Layout form components are able to save child data to relationships, such as **BelongsTo**:

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

Fieldset::make('Author')
->relationship('author')
->schema([
    TextInput::make('name')->required(),
    Textarea::make('bio'),
])
```

For more information, see the [Form docs](#).

HasOne**Layout component**

Layout form components are able to save child data to relationships, such as **HasOne**:

```
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'),
])
```

For more information, see the [Form docs](#).

HasMany**Relation manager**

"Relation managers" in Filament allow admins to list, create, associate, edit, dissociate and delete related records without leaving the resource's Edit page.

The related records are listed in a table, which has buttons to open a modal for each action.

For more information on relation managers, see the [full documentation](#).

Repeater

Alternatively, if you're looking to edit the relationship from the main form, you could use a [repeater](#):

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
```

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.

HasManyThrough

Relation manager

"Relation managers" in Filament allow admins to list, create, edit, and delete related records without leaving the resource's Edit page.

The related records are listed in a table, which has buttons to open a modal for each action.

For more information on relation managers, see the [full documentation](#).

BelongsToMany

Multi-select field

Filament can automatically load [Select](#) options from a [BelongsToMany](#) relationship:

```
use Filament\Forms\Components\Select;

Select::make('technologies')
    ->multiple()
    ->relationship('technologies', 'name')
```

More information is available in the [Form docs](#).

Checkbox list field

Filament can automatically load [CheckboxList](#) options from a [BelongsToMany](#) relationship:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
    ->relationship('technologies', 'name')
```

More information about [CheckboxList](#) is available in the [Form docs](#).

Relation manager

"Relation managers" in Filament allow admins to list, create, attach, edit, detach and delete related records without leaving the resource's Edit page.

The related records are listed in a table, which has buttons to open a modal for each action.

For more information on relation managers, see the [full documentation](#).

MorphTo

Select field

Filament includes the ability to automatically load options from a `MorphTo` relationship:

```
use Filament\Forms\Components\MorphToSelect;

MorphToSelect::make('commentable')
->types([
    MorphToSelect\Type::make(Product::class)->titleColumnName('name'),
    MorphToSelect\Type::make(Post::class)->titleColumnName('title'),
])
```

More information is available in the [Form docs](#).

MorphOne

Layout component

Layout form components are able to [save child data to relationships](#), such as `MorphOne`:

```
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'),
])
```

For more information, see the [Form docs](#).

MorphMany

Relation manager

"Relation managers" in Filament allow admins to list, create, associate, edit, dissociate and delete related records without leaving the resource's Edit page.

The related records are listed in a table, which has buttons to open a modal for each action.

For more information on relation managers, see the [full documentation](#).

Repeater

Alternatively, if you're looking to edit the relationship from the main form, you could use a [repeater](#):

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
->schema([
    // ...
])
```

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.

MorphToMany

Relation manager

"Relation managers" in Filament allow admins to list, create, attach, edit, detach and delete related records without leaving the resource's Edit page.

The related records are listed in a table, which has buttons to open a modal for each action.

For more information on relation managers, see the [full documentation](#).

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](#).

Model labels

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');
}
```

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');
}
```

Navigation

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');
}
```

Icons

The `$navigationIcon` property supports the name of any Blade component. By default, the Blade Heroicons v1 package is installed, so you may use the name of any Heroicons v1 out of the box. 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:

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

Sorting 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 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:

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

Customizing the 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 [model scopes](#) that affect the entire resource. You can use this to implement [multi-tenancy](#) easily within the admin panel.

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 URL slug

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

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

Multi-tenancy

Multi-tenancy, simply, is the concept of users "owning" records, and only being able to access the records that they own within Filament.

Simple multi-tenancy from scratch

Simple multi-tenancy is easy to set up with Filament.

First, scope the base Eloquent query for every "owned" resource by defining the `getEloquentQuery()` method:

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

In this example we use `whereBelongsTo()` to scope the records to only those that belong to the currently authenticated user. However, you may use whatever Eloquent method you wish here, including a manual `where()` clause, or a scope.

Finally, you need to ensure that records are attached to the current user when they are first created. The easiest way to do this is through a model observer:

```
public function creating(Post $post): void
{
    $post->user() -> associate(auth() -> user());
}
```

Additionally, you may want to scope the options available in the relation manager `AttachAction` or `AssociateAction`:

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

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

stancl/tenancy

To set up `stancl/tenancy` to work with Filament, you just need to add the `InitializeTenancyByDomain::class` middleware to Livewire and the Filament config file:

```
use Stancl\Tenancy\Middleware\InitializeTenancyByDomain;

'middleware' => [
    // ...
    'base' => [
        // ...
        InitializeTenancyByDomain::class
    ],
],
]
```

Deleting 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'),
    ];
}
```

Listing Records

Columns

The `$table->columns()` method is used to define the columns in your table. It is an array of column objects, in the order they should appear in your table.

We have many columns available for your tables, including:

- [Text column](#)
- [Image column](#)
- [Icon column](#)
- [Badge column](#)

To view a full list of available table columns, see the [Table Builder documentation](#).

You may also build your own completely custom table columns.

Sorting a column by default

If a column is `sortable()`, you may choose to sort it by default using the `$table->defaultSort()` method:

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

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

Filters

Filters are predefined scopes that administrators can use to filter records in your table. The `$table->filters()` method is used to register these.

Displaying filters above or below the table content

To render the filters above the table content instead of in a popover, you may use:

```
use Filament\Tables\Filters\Layout;
use Filament\Resources\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters(
            [
                // ...
            ],
            layout: Layout::AboveContent,
        );
}
```

To render the filters below the table content instead of in a popover, you may use:

```
use Filament\Tables\Filters\Layout;
use Filament\Resources\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters(
            [
                // ...
            ],
            layout: Layout::BelowContent,
        );
}
```

To render the filters above the table content in a collapsible panel, you may use:

```
use Filament\Tables\Filters\Layout;
use Filament\Resources\Table;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->filters(
            [
                // ...
            ],
            layout: Layout::AboveContentCollapsible,
        );
}
```

Actions

Actions are buttons that are rendered at the end of table rows. They allow the user to perform a task on a record in the table. To learn how to build actions, see the [full actions documentation](#).

To add actions to a table, use the `$table->actions()` method:

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

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->actions([
            // ...
            Tables\Actions\Action::make('activate')
                ->action(fn (Post $record) => $record->activate())
                ->requiresConfirmation()
                ->color('success'),
        ]);
}
```

Grouping actions

You may use an `ActionGroup` object to group multiple table actions together in a dropdown:

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

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

Bulk actions

Bulk actions are buttons that are rendered in a dropdown in the header of the table. They appear when you select records using the checkboxes at the start of each table row. They allow the user to perform a task on multiple records at once in the table. To learn how to build bulk actions, see the [full actions documentation](#).

To add bulk actions, use the `$table->bulkActions()` method:

```
use Filament\Resources\Table;
use Filament\Tables;
use Illuminate\Database\Eloquent\Collection;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->bulkActions([
            // ...
            Tables\Actions\BulkAction::make('activate')
                ->action(fn (Collection $records) => $records->each->activate())
                ->requiresConfirmation()
                ->color('success')
                ->icon('heroicon-o-check'),
        ]);
}
```

Record select checkbox position

By default, the record select checkboxes are rendered at the start of the row. You may move them to the end of the row:

```
use Filament\Resources\Table;
use Filament\Tables\Actions\RecordCheckboxPosition;

public static function table(Table $table): Table
{
    return $table
        ->recordCheckboxPosition(RecordCheckboxPosition::AfterCells)
        ->columns([
            // ...
        ])
        ->bulkActions([
            // ...
        ]);
}
```

Reordering records

To allow the user to reorder records using drag and drop in your table, you can use the `reorderable()` method:

```
use Filament\Resources\Table;

public static 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. Pagination will be disabled in reorder mode to allow you to move records between pages.

The `reorderable()` method passes in 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 pass this in to `reorderable()`:

```
use Filament\Resources\Table;

public static function table(Table $table): Table
{
    return $table
        // ...
        ->reorderable('order_column');
}
```

Polling content

You may poll table content so that it refreshes at a set interval, using the `poll()` method:

```
use Filament\Resources\Table;

public static function table(Table $table): Table
{
    return $table
        // ...
        ->poll('10s');
}
```

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 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, override the `getTableQuery()` method on the page class:

```
protected function getTableQuery(): Builder
{
    return parent::getTableQuery()->withoutGlobalScopes();
}
```

Customizing the query string

Table search, filters, sorts and other stateful properties are stored in the URL as query strings. Since Filament uses Livewire internally, this behaviour can be modified by overriding the `$queryString` property on the List page of the resource. For instance, you can employ `query string aliases` to rename some of the properties using `as`:

```
protected $queryString = [
    'isTableReordering' => ['except' => false],
    'tableFilters' => ['as' => 'filters'], // `tableFilters` is now replaced with `filters` in
    the query string
    'tableSortColumn' => ['except' => ''],
    'tableSortDirection' => ['except' => ''],
    'tableSearchQuery' => ['except' => '', 'as' => 'search'], // `tableSearchQuery` is now
replaced with `search` in the query string
];
```

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.list-users';
```

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:

```
use Filament\Pages\Actions\CreateAction;

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 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:

```
use Filament\Pages\Actions\CreateAction;
use Illuminate\Database\Eloquent\Model;

CreateAction::make()
->using(function (array $data): Model {
    return static::getModel()::create($data);
})
```

Customizing form 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:

```
use Filament\Pages\Actions\CreateAction;

CreateAction::make()
    ->successNotificationTitle('User registered')
```

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.');
}
```

Alternatively, if you're creating records in a modal action:

```
use Filament\Notifications\Notification;
use Filament\Pages\Actions\CreateAction;

CreateAction::make()
    ->successNotification(
        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;
}
```

Alternatively, if you're creating records in a modal action:

```
use Filament\Pages\Actions\CreateAction;

CreateAction::make()
    ->successNotification(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:

```
use Filament\Pages\Actions\CreateAction;

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

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

Alternatively, if you're creating records in a modal action:

```

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

CreateAction::make()
    ->before(function (CreateAction $action) {
        if (! $this->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();
```

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`.

Wizards

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

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()
                    ->reactive()
                    ->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, simply define a `steps()` array and pass your `Step` objects:

```

use Filament\Pages\Actions\CreateAction;

CreateAction::make()
    ->steps([
        // ...
    ])

```

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;
}

```

Alternatively, if you're creating records in a modal action:

```
use Filament\Pages\Actions\CreateAction;

CreateAction::make()
    ->steps([
        // ...
    ])
    ->skippableSteps()
```

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\Resources\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()
            ->reactive()
            ->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(),
                ]),
            // ...
        ];
    }
}
```

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';
```

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:

```
use Filament\Tables\Actions>EditAction;

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 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:

```
use Filament\Tables\Actions>EditAction;

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 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:

```
use Filament\Tables\Actions>EditAction;
use Illuminate\Database\Eloquent\Model;

EditAction::make()
    ->using(function (Model $record, array $data): Model {
        $record->update($data);

        return $record;
    })
}
```

Customizing form 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->record]);
```

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:

```
use Filament\Tables\Actions>EditAction;

EditAction::make()
->successNotificationTitle('User updated')
```

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.');
}
```

Alternatively, if you're editing records in a modal action:

```
use Filament\Notifications\Notification;
use Filament\Tables\Actions>EditAction;

EditAction::make()
->successNotification(
    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;
}
```

Alternatively, if you're editing records in a modal action:

```
use Filament\Tables\Actions>EditAction;

EditAction::make()
->successNotification(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:

```
use Filament\Tables\Actions>EditAction;

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 `$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->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();
    }

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

Alternatively, if you're editing records in a modal action:

```

use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;
use Filament\Tables\Actions>EditAction;

EditAction::make()
    ->before(function (EditAction $action) {
        if (! $this->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();
```

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 that runs the `impersonate()` Livewire method:

```

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

class EditUser extends EditRecord
{
    // ...

    protected function getActions(): array
    {
        return [
            Actions\Action::make('impersonate')->action('impersonate'),
            Actions\DeleteAction::make(),
        ];
    }

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

```

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

```

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

class EditUser extends EditRecord
{
    // ...

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

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

```

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

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';

```

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
```

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\Resources\Table;
use Filament\Tables;

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:

```
use Filament\Tables\Actions\ViewAction;

ViewAction::make()
->mutateRecordDataUsing(function (array $data): array {
    $data['user_id'] = auth()->id();

    return $data;
})
```

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`.

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';
```

Deleting Records

Handling soft deletes

Creating a resource with soft deletes

By default, you will not be able to interact with deleted records in the admin panel. 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\Resources\Table;
use Filament\Tables;
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\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\Pages\Actions;

protected function getActions(): 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\Resources\Table;
use Filament\Tables;

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

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 () {
    // ...
})
```

Halting the deletion process

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

```

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

DeleteAction::make()
    ->before(function (DeleteAction $action) {
        if (! $this->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();
```

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

Getting started

"Relation managers" in Filament allow administrators to list, create, attach, associate, edit, detach, dissociate and delete related records without leaving the resource's Edit page. Resource classes contain a static `getRelations()` method that is used to register relation managers for your resource.

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 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\Resources\Form;
use Filament\Resources\Table;
use Filament\Tables;

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

public static 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,
    ];
}
```

For relationships with unconventional naming conventions, you may wish to include the `$inverseRelationship` property on the relation manager:

```
protected static ?string $inverseRelationship = 'section'; // Since the inverse related model is
`Category`, this is normally `category`, not `section`.
```

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.

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 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.

As per the documentation on [listing records](#), you may use all the same utilities for customization on the relation manager:

- [Columns](#)
- [Filters](#)
- [Actions](#)
- [Bulk actions](#)

Additionally, you may use any other feature of the [table builder](#).

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\Forms;
use Filament\Resources\Form;
use Filament\Tables;

public static 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 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\Resources\Form;
use Filament\Tables;

public static 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 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 accepts the `$data` as an array, and returns the modified version:

```
use Filament\Tables\Actions\CreateAction;

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 using the `using()` method:

```
use Filament\Tables\Actions\CreateAction;
use Filament\Tables\Contracts\HasRelationshipTable;
use Illuminate\Database\Eloquent\Model;

CreateAction::make()
    ->using(function (HasRelationshipTable $livewire, array $data): Model {
        return $livewire->getRelationship()->create($data);
    })
}
```

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 text content of this notification:

```
use Filament\Tables\Actions\CreateAction;

CreateAction::make()
    ->successNotificationTitle('User registered')
```

And to disable the notification altogether:

```
use Filament\Tables\Actions\CreateAction;

CreateAction::make()
    ->successNotification(null)
```

Lifecycle hooks

Hooks may be used to execute code at various points within an action's lifecycle.

```
use Filament\Tables\Actions\CreateAction;

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 Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;
use Filament\Resources\RelationManagers\RelationManager;
use Filament\Tables\Actions\CreateAction;

CreateAction::make()
->before(function (CreateAction $action, RelationManager $livewire) {
    if (! $livewire->ownerRecord->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();
```

Editing 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\Resources\Form;
use Filament\Tables;

public static 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 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:

```
use Filament\Tables\Actions>EditAction;

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 define a `mutateFormDataUsing()` method, which accepts the `$data` as an array, and returns it modified:

```
use Filament\Tables\Actions>EditAction;

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 using the `using()` method:

```
use Filament\Tables\Actions>EditAction;
use Illuminate\Database\Eloquent\Model;

EditAction::make()
->using(function (Model $record, array $data): Model {
    $record->update($data);

    return $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 text content of this notification:

```
use Filament\Tables\Actions>EditAction;

EditAction::make()
->successNotificationTitle('User updated')
```

And to disable the notification altogether:

```
use Filament\Tables\Actions>EditAction;

EditAction::make()
->successNotification(null)
```

Lifecycle hooks

Hooks may be used to execute code at various points within an action's lifecycle.

```
use Filament\Tables\Actions>EditAction;

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 Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;
use Filament\Resources\RelationManagers\RelationManager;
use Filament\Tables\Actions>EditAction;

EditAction::make()
    ->before(function (EditAction $action, RelationManager $livewire) {
        if (! $livewire->ownerRecord->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();
```

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\Resources\Table;
use Filament\Tables;

public static function table(Table $table): Table
{
    return $table
        ->columns([
            // ...
        ])
        ->headerActions([
            // ...
            Tables\Actions\AttachAction::make(),
        ])
        ->actions([
            // ...
            Tables\Actions\DetachAction::make(),
        ])
        ->bulkActions([
            // ...
            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()` outputs 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

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()) )
```

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 `$allowsDuplicates` property to the relation manager:

```
protected bool $allowsDuplicates = true;
```

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\Resources\Table;
use Filament\Tables;

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

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()) )
```

Viewing 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\Resources\Table;
use Filament\Tables;

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

Deleting 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\Resources\Table;
use Filament\Tables;
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([
            Tables\Actions\DeleteAction::make(),
            Tables\Actions\ForceDeleteAction::make(),
            Tables\Actions\RestoreAction::make(),
            // ...
        ])
        ->bulkActions([
            Tables\Actions\DeleteBulkAction::make(),
            Tables\Actions\ForceDeleteBulkAction::make(),
            Tables\Actions\RestoreBulkAction::make(),
            // ...
        ]);
}

protected function getTableQuery(): Builder
{
    return parent::getTableQuery()
        ->withoutGlobalScopes([
            SoftDeletingScope::class,
        ]);
}

```

Lifecycle hooks

You can use the `before()` and `after()` methods to execute code before and after a record is deleted:

```

use Filament\Tables\Actions\DeleteAction;

DeleteAction::make()
    ->before(function () {
        // ...
    })
    ->after(function () {
        // ...
    })

```

Halting the deletion process

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

```

use Filament\Notifications\Actions\Action;
use Filament\Notifications\Notification;
use Filament\Resources\RelationManagers\RelationManager;
use Filament\Tables\Actions\DeleteAction;

DeleteAction::make()
    ->before(function (DeleteAction $action, RelationManager $livewire) {
        if (! $livewire->ownerRecord->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();
```

Accessing the 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 mounted in a public `$ownerRecord` property. Thus, you may access the owner record using:

```
$this->ownerRecord
```

However, in 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\Resources\Form;
use Filament\Resources\RelationManagers\RelationManager;

public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\Select::make('store_id')
                ->options(function (RelationManager $livewire): array {
                    return $livewire->ownerRecord->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,
        ]),
        // ...
    ];
}

```

Conditional visibility

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:

```

use Illuminate\Database\Eloquent\Model;

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

```

Moving the resource form to tabs

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

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

Global Search

Global search allows you to search across all of your resource records, from anywhere in the admin panel.

Title

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.

Note: 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:

```
public static function getGlobalSearchResultTitle(Model $record): string
{
    return $record->name;
}
```

Multi-column search

If you would like to search across multiple columns of your resource, you may override the `getGloballySearchableAttributes()` method. "Dot-syntax" allows you to search inside of relationships:

```
public static function getGloballySearchableAttributes(): array
{
    return ['title', 'slug', 'author.name', 'category.name'];
}
```

Details

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:

```
protected static function getGlobalSearchEloquentQuery(): Builder
{
    return parent::getGlobalSearchEloquentQuery()->with(['author', 'category']);
}
```

URL

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]);
}
```

Actions

Global search support actions that render a button or link which may open a URL or emit a Livewire event. Actions will render as link by default, but you may configure it to render a button using the `button()` method.

Actions can be defined as follows:

```
use Filament\GlobalSearch\Actions\Action;

public static function getGlobalSearchResultActions(Model $record): array
{
    return [
        Action::make('edit')
            ->iconButton()
            ->icon('heroicon-s-pencil')
            ->url(static::getUrl('edit', ['record' => $record])),
    ];
}
```

Widgets

Getting started

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

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;
```

Custom Pages

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'),
    ];
}
```

Please note that the order of pages registered in this method matters - any wildcard route segments that are defined before hard-coded 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](#).

To generate a URL for a resource route, you may call the static `getUrl()` method on the resource class:

```
UserResource::getUrl('sort');
```

Security

Protecting model attributes

Filament will expose all model attributes to JavaScript, except if they are `$hidden` on your model. This is Livewire's behaviour 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

Getting Started

Filament allows you to create completely custom pages for the admin panel.

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 admin panel.

Conditionally hiding pages in navigation

You can prevent pages from appearing in the menu by overriding the `shouldRegisterNavigation()` method in your Page class. This is useful if you want to control which users can see the page in the sidebar.

```
protected static function shouldRegisterNavigation(): bool
{
    return auth() -> user() -> canManageSettings();
}
```

Please be aware that all users will still be able to visit this page through its direct URL, so to fully limit access you must also check in the `mount()` method of the page:

```
public function mount(): void
{
    abort_unless(auth() -> user() -> canManageSettings(), 403);
}
```

Customization

Filament will automatically generate a title, navigation label and URL (slug) for your page based on its name. You may override it using static properties of your page class:

```
protected static ?string $title = 'Custom Page Title';

protected static ?string $navigationLabel = 'Custom Navigation Label';

protected static ?string $slug = 'custom-url-slug';
```

You may also specify a custom header and footer view for any page. You may return them from the `getHeader()` and `getFooter()` methods:

```
use Illuminate\Contracts\View\View;

protected function getHeader(): View
{
    return view('filament.settings.custom-header');
}

protected function getFooter(): View
{
    return view('filament.settings.custom-footer');
}
```

Actions

Getting started

"Actions" are buttons that are displayed next to the page's heading, and allow the user to run a Livewire method on the page or visit a URL.

To define actions for a page, use the `getActions()` method:

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')->action('openSettingsModal'),
    ];
}

public function openSettingsModal(): void
{
    $this->dispatchBrowserEvent('open-settings-modal');
}
```

The button's label is generated based on its name. To override it, you may use the `label()` method:

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')
            ->label('Settings')
            ->action('openSettingsModal'),
    ];
}
```

You may also allow the button to open a URL, using the `url()` method:

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')
            ->label('Settings')
            ->url(route('settings')),
    ];
}
```

Buttons may have a `color()`. The default is `primary`, but you may use `secondary`, `success`, `warning`, or `danger`:

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')->color('secondary'),
    ];
}
```

Buttons may have a `size()`. The default is `md`, but you may also use `sm` or `lg`:

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')->size('lg'),
    ];
}
```

Buttons may also have an `icon()`, which is the name of any Blade component. By default, the [Blade Heroicons v1](#) package is installed, so you may use the name of any [Heroicons v1](#) out of the box. However, you may create your own custom icon components or install an alternative library if you wish.

```
use Filament\Pages\Actions\Action;

protected function getActions(): array
{
    return [
        Action::make('settings')->icon('heroicon-s-cog'),
    ];
}
```

Modals

Actions may require additional confirmation or form information 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 Filament\Pages\Actions\Action;

Action::make('delete')
    ->action(fn () => $this->record->delete())
    ->requiresConfirmation()
```

Note: 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()` callback.

Custom forms

You may also render a form in this 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()` callback:

```
use App\Models\User;
use Filament\Forms;
use Filament\Pages\Actions\Action;

Action::make('updateAuthor')
    ->action(function (array $data): void {
        $this->record->author()->associate($data['authorId']);
        $this->record->save();
    })
    ->form([
        Forms\Components\Select::make('authorId')
            ->label('Author')
            ->options(User::query()->pluck('name', 'id'))
            ->required(),
    ])
])
```

Filling default data

You may fill the form with default data, using the `mountUsing()` method:

```
use App\Models\User;
use Filament\Forms;
use Filament\Pages\Actions\Action;

Action::make('updateAuthor')
    ->mountUsing(fn (Forms\ComponentContainer $form) => $form->fill([
        'authorId' => $this->record->author->id,
    ]))
    ->action(function (array $data): void {
        $this->record->author()->associate($data['authorId']);
        $this->record->save();
    })
    ->form([
        Forms\Components\Select::make('authorId')
            ->label('Author')
            ->options(User::query()->pluck('name', 'id'))
            ->required(),
    ])
])
```

Setting a modal heading, subheading, and button label

You may customize the heading, subheading and button label of the modal:

```
use Filament\Pages\Actions\Action;

Action::make('delete')
    ->action(fn () => $this->record->delete())
    ->requiresConfirmation()
    ->modalHeading('Delete posts')
    ->modalSubheading('Are you sure you\'d like to delete these posts? This cannot be undone.')
    ->modalButton('Yes, delete them')
```

Custom 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 Filament\Pages\Actions\Action;

Action::make('advance')
    ->action(fn () => $this->record->advance())
    ->modalContent(view('filament.pages.actions.advance'))
```

By default, the custom content is displayed above the modal form if there is one, but you can add content below using `modalFooter()` if you wish:

```
use Filament\Pages\Actions\Action;

Action::make('advance')
    ->action(fn () => $this->record->advance())
    ->modalFooter(view('filament.pages.actions.advance'))
```

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()`:

```
use Filament\Pages\Actions\Action;

Action::make('create')
    ->action('create')
    ->modalHidden(fn (): bool => $this->role !== 'admin')
    ->modalContent(view('filament.pages.actions.create'))
```

Grouping

You may use an `ActionGroup` object to group multiple actions together in a dropdown:

```
use Filament\Pages\Actions;

protected function getActions(): array
{
    return [
        Actions\ActionGroup::make([
            Actions\ViewAction::make(),
            Actions>EditAction::make(),
            Actions>DeleteAction::make(),
        ]),
    ];
}
```

Keybindings

You can attach keyboard shortcuts to actions. These use the same key codes as [Mousetrap](#):

```
use Filament\Pages\Actions\Action;

Action::make('save')
->action(fn () => $this->save())
->keyBindings(['command+s', 'ctrl+s'])
```

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 Filament\Pages\Actions\Action;

Action::make('approve')
->action(function () {
    $this->record->approve();

    $this->refreshFormData([
        'status',
    ]);
})
```

This method accepts an array of model attributes that you wish to refresh in the form.

Widgets

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:

```
protected 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:

```
protected function getHeaderWidgetsColumns(): int | array
{
    return [
        'md' => 4,
        'xl' => 5,
    ];
}
```

This pairs well with [responsive widget widths](#).

Dashboard

Getting Started

Filament allows you to build dynamic dashboards, comprised of "widget" cards, very easily.

Widgets are pure [Livewire](#) components, so may use any features of that package.

Widgets may also be used on [resource pages](#) or other [custom pages](#).

Available widgets

Filament ships with a few pre-built widgets, as well as the ability to create [custom widgets](#):

- [Stats](#) widgets display any data, often numeric data, within cards in a row.
- [Chart](#) widgets display numeric data in a visual chart.
- [Table](#) widgets render data in a table, which supports sorting, searching, filtering, actions, and everything else included within the [table builder](#).

You may also [create your own custom 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:

```
protected function getColumns(): 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:

```
protected function getColumns(): int | 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();
}
```

Disabling the default widgets

By default, two widgets are displayed on the dashboard. These widgets can be disabled by updating the `widgets.register` property of the `configuration` file:

```
'widgets' => [
    // ...
    'register' => [],
],
```

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.

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;

use Filament\Pages\Dashboard as BasePage;

class Dashboard extends BasePage
{
    // ...
}
```

Finally, remove the original `Dashboard` class from the [configuration file](#):

```
'pages' => [
    // ...
    'register' => [],
],
```

Stats

Getting started

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
```

Then return `Card` instances from the `getCards()` method:

```
<?php

namespace App\Filament\widgets;

use Filament\widgets\StatsOverviewWidget as BaseWidget;
use Filament\widgets\StatsOverviewWidget\Card;

class StatsOverviewWidget extends BaseWidget
{
    protected function getCards(): array
    {
        return [
            Card::make('Unique views', '192.1k'),
            Card::make('Bounce rate', '21%'),
            Card::make('Average time on page', '3:12'),
        ];
    }
}
```

Now, check out your widget in the dashboard.

Card descriptions and icons

You may add a `description()` to provide additional information, along with a `descriptionIcon()`:

```

protected function getCards(): array
{
    return [
        Card::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-s-trending-up'),
        Card::make('Bounce rate', '21%')
            ->description('7% increase')
            ->descriptionIcon('heroicon-s-trending-down'),
        Card::make('Average time on page', '3:12')
            ->description('3% increase')
            ->descriptionIcon('heroicon-s-trending-up'),
    ];
}

```

Card colors

You may also give cards a `color()` ([`primary`], [`success`], [`warning`] or [`danger`]):

```

protected function getCards(): array
{
    return [
        Card::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-s-trending-up')
            ->color('success'),
        Card::make('Bounce rate', '21%')
            ->description('7% increase')
            ->descriptionIcon('heroicon-s-trending-down')
            ->color('danger'),
        Card::make('Average time on page', '3:12')
            ->description('3% increase')
            ->descriptionIcon('heroicon-s-trending-up')
            ->color('success'),
    ];
}

```

Card extra HTML attributes

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

```

protected function getCards(): array
{
    return [
        Card::make('Processed', '192.1k')
            ->color('success')
            ->extraAttributes([
                'class' => 'cursor-pointer',
                'wire:click' => '$emitUp("setStatusFilter", "processed")',
            ]),
            // ...
    ];
}

```

Card charts

You may also add or chain a `chart()` to each card to provide historical data. The `chart()` method accepts an array of data points to plot:

```
protected function getCards(): array
{
    return [
        Card::make('Unique views', '192.1k')
            ->description('32k increase')
            ->descriptionIcon('heroicon-s-trending-up')
            ->chart([7, 2, 10, 3, 15, 4, 17])
            ->color('success'),
        // ...
    ];
}
```

Live updating (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;
```

Charts

Getting started

Filament comes with many "chart" widget template, 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 are several chart classes available, but we'll use the `LineChartWidget` class for this example.

The `getHeading()` method is used to return a heading that describes the chart.

The `getData()` method is used to return an array of datasets and labels. Each dataset is a labelled array of points to plot on the chart, and each label is a string. This structure is identical with 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\LineChartWidget;

class BlogPostsChart extends LineChartWidget
{
    protected function getHeading(): string
    {
        return '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'],
        ];
    }
}
```

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 what to return from `getData()`:

- `Filament\Widgets\BarChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\BubbleChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\DoughnutChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\LineChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\PieChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\PolarAreaChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\RadarChartWidget` - [Chart.js documentation](#)
- `Filament\Widgets\ScatterChartWidget` - [Chart.js documentation](#)

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 (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 `LineChartWidget` class:

```
protected static ?array $options = [
    'plugins' => [
        'legend' => [
            'display' => false,
        ],
    ],
];
```

Tables

Filament comes with a "table" widget template, which you can use to display a table of data without needing to write a custom view.

Start by creating a widget with the command:

```
php artisan make:filament-widget LatestOrders --table
```

Then update the `getTableQuery()` and `getTableColumns()` methods to return the data query and columns you want to display:

```
<?php

namespace App\Filament\Widgets;

use App\Models\Order;
use Closure;
use Filament\Tables;
use Filament\Widgets\TableWidget as BaseWidget;
use Illuminate\Database\Eloquent\Builder;

class LatestOrders extends BaseWidget
{
    protected function getTableQuery(): Builder
    {
        return Order::query()->latest();
    }

    protected function getTableColumns(): array
    {
        return [
            Tables\Columns\TextColumn::make('id'),
            Tables\Columns\TextColumn::make('customer.name')
                ->label('Customer'),
        ];
    }
}
```

Now, check out your widget in the dashboard.

Table widgets support all features of the [Table Builder](#), including [filters](#) and [actions](#).

Navigation

Getting started

By default, Filament will register navigation items for each of your [resources](#) and [custom pages](#). These classes contain static properties and methods that you can override, to configure that navigation item and its order:

```
protected static ?string $navigationIcon = 'heroicon-o-document-text';

protected static ?string $navigationLabel = 'Custom Navigation Label';

protected static ?int $navigationSort = 3;
```

The `$navigationIcon` supports the name of any Blade component, and passes a set of formatting classes to it. By default, the [Blade Heroicons v1](#) package is installed, so you may use the name of any [Heroicons v1](#) out of the box. However, you may create your own custom icon components or install an alternative library if you wish.

Navigation item badges

To add a badge next to the navigation item, you can use the `getNavigationBadge()` method and return the content of the badge:

```
protected static function getNavigationBadge(): ?string
{
    return static::getModel()::count();
}
```

If a badge value is returned by `getNavigationBadge()`, it will display using the primary Tailwind color by default. To style the badge contextually, return either `danger`, `warning`, `success` or `secondary` from the `getNavigationBadgeColor()` method:

```
protected static function getNavigationBadgeColor(): ?string
{
    return static::getModel()::count() > 10 ? 'warning' : 'primary';
}
```

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 top of the sidebar.

Customizing navigation groups

You may customize navigation groups by calling `Filament::registerNavigationGroups()` from the `boot()` method of any service provider, and passing `NavigationGroup` objects in order:

```

use Filament\Facades\Filament;
use Filament\Navigation\NavigationGroup;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Filament::serving(function () {
            Filament::registerNavigationGroups([
                NavigationGroup::make()
                    ->label('Shop')
                    ->icon('heroicon-s-shopping-cart'),
                NavigationGroup::make()
                    ->label('Blog')
                    ->icon('heroicon-s-pencil'),
                NavigationGroup::make()
                    ->label('Settings')
                    ->icon('heroicon-s-cog')
                    ->collapsed(),
            ]);
        });
    }
}

```

In this example, we pass in a custom `icon()` for the groups, and make one `collapsed()` by default.

Ordering navigation groups

By using `registerNavigationGroups()`, you are defining a new order for the navigation groups in the sidebar. 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:

```

use Filament\Facades\Filament;

Filament::registerNavigationGroups([
    'Shop',
    'Blog',
    'Settings',
]);

```

Active icons

You may assign a navigation icon which will be displayed for active items using the `$activeNavigationIcon` property:

```
protected static ?string $activeNavigationIcon = 'heroicon-s-document-text';
```

Alternatively, override the `getActiveNavigationIcon()` method:

```

protected static function getActiveNavigationIcon(): string
{
    return 'heroicon-s-document-text';
}

```

Registering custom navigation items

You may register custom navigation items by calling `Filament::registerNavigationItems()` from the `boot()` method of any service provider:

```
use Filament\Facades\Filament;
use Filament\Navigation\NavigationItem;

Filament::serving(function () {
    Filament::registerNavigationItems([
        NavigationItem::make('Analytics')
            ->url('https://filament.pirsch.io', shouldOpenInNewTab: true)
            ->icon('heroicon-o-presentation-chart-line')
            ->activeIcon('heroicon-s-presentation-chart-line')
            ->group('Reports')
            ->sort(3),
    ]);
});
```

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(auth() ->user() ->can('view-analytics'))
    // or
    ->hidden(! 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;
```

Advanced navigation customization

The `Filament::navigation()` method which can be called from the `boot()` method of a `ServiceProvider`:

```
use Filament\Facades\Filament;
use Filament\Navigation\NavigationBuilder;

Filament::navigation(function (NavigationBuilder $builder): NavigationBuilder {
    return $builder;
});
```

Once you add this callback function, Filament's default automatic navigation will be disabled and your sidebar will be empty. This is done on purpose, since this API is designed to give you complete control over the navigation.

To register navigation items, just call the `items()` method:

```

use App\Filament\Pages\Settings;
use App\Filament\Resources\UserResource;
use Filament\Facades\Filament;
use Filament\Navigation\NavigationBuilder;
use Filament\Navigation\NavigationItem;

Filament::navigation(function (NavigationBuilder $builder): NavigationBuilder {
    return $builder->items([
        NavigationItem::make('Dashboard')
            ->icon('heroicon-o-home')
            ->activeIcon('heroicon-s-home')
            ->isActiveWhen(fn (): bool => request()->routeIs('filament.pages.dashboard'))
            ->url(route('filament.pages.dashboard')),
            ...UserResource::getNavigationItems(),
            ...Settings::getNavigationItems(),
    ]);
});

```

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\Facades\Filament;
use Filament\Navigation\NavigationBuilder;
use Filament\Navigation\NavigationGroup;

Filament::navigation(function (NavigationBuilder $builder): NavigationBuilder {
    return $builder
        ->groups([
            NavigationGroup::make('Website')
                ->items([
                    ...PageResource::getNavigationItems(),
                    ...CategoryResource::getNavigationItems(),
                    ...HomePageSettings::getNavigationItems(),
                ]),
        ]);
});

```

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 should use a service provider:

```
use Filament\Facades\Filament;
use Filament\Navigation\UserMenuItem;

Filament::serving(function () {
    Filament::registerUserMenuItems([
        UserMenuItem::make()
            ->label('Settings')
            ->url(route('filament.pages.settings'))
            ->icon('heroicon-s-cog'),
        // ...
    ]);
});
```

Customizing the account link

To customize the user account link at the start of the user menu, register a new item with the `account` array key:

```
use Filament\Facades\Filament;
use Filament\Navigation\UserMenuItem;

Filament::serving(function () {
    Filament::registerUserMenuItems([
        'account' => UserMenuItem::make()->url(route('filament.pages.account')),
        // ...
    ]);
});
```

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\Facades\Filament;
use Filament\Navigation\UserMenuItem;

Filament::serving(function () {
    Filament::registerUserMenuItems([
        // ...
        'logout' => UserMenuItem::make()->label('Log out'),
    ]);
});
```

Users

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 admin panel.

Authorizing access to the admin 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 Illuminate\Foundation\Auth\User as Authenticatable;

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

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

The `canAccessFilament()` method returns `true` or `false` depending on whether the user is allowed to access Filament. In this example, we check if the user's email ends with `@yourdomain.com` and if they have verified their email address.

Setting up avatars

Out of the box, Filament uses [ui-avatars.com](#) to generate avatars based on a user's name. To provide your own avatar URLs, 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](#).

Configuring the name attribute

By default, Filament will use the `name` attribute of the user to display their name in the admin panel. 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.

Notifications

Sending notifications

The admin panel uses the [Notifications](#) package to send messages to users.

Please read the [documentation](#) to discover how to send notifications easily.

However, there are a few differences in configuration when using the admin panel.

Database notifications

Instead of enabling database notifications inside the notifications package, you must enable it for the admin panel specifically.

First, you must [publish the configuration file](#) for the admin panel.

Now, enable database notifications:

```
'database_notifications' => [
    'enabled' => true,
    // ...
],
```

You may also control [polling](#):

```
'database' => [
    'enabled' => true,
    'polling_interval' => '30s',
    // ...
],
```

Echo

Some features of the notifications package, including [receiving real-time database notifications](#) and [broadcast notifications](#), require Laravel Echo to be installed.

Firstly, you must set up a [server-side websockets integration](#) like Pusher.

Then, define your Echo configuration within the admin panel [configuration file](#):

```
'broadcasting' => [
    'echo' => [
        'broadcaster' => 'pusher',
        'key' => env('VITE_PUSHER_APP_KEY'),
        'cluster' => env('VITE_PUSHER_APP_CLUSTER'),
        'forceTLS' => true,
    ],
],
```

Appearance

Changing the brand logo

By default, Filament will use your app's name as a brand logo in the admin panel.

You may create a `resources/views/vendor/filament/components/brand.blade.php` file to provide a custom logo:

```

```

Dark mode

By default, Filament only includes a light theme. However, you may allow the user to switch to dark mode if they wish, using the `dark_mode` setting of the [configuration file](#):

```
'dark_mode' => true,
```

When dark mode is enabled, the admin panel will automatically obey your system's dark / light mode preference. You may switch to dark / light mode permanently through the button in the user dropdown menu.

If you're using a [custom theme](#), make sure that you have the `darkMode: 'class'` setting in your `tailwind.config.js` file.

Please note: before enabling dark mode in production, please thoroughly test your admin panel - especially third party plugins, which may not be properly tested with dark mode.

When the user toggles between dark or light mode, a browser event called `dark-mode-toggled` is dispatched. You can listen to it:

```
<div
    x-data="{ mode: 'light' }"
    x-on:dark-mode-toggled.window="mode = $event.detail"
>
    <span x-show="mode === 'light'">
        Light mode
    </span>

    <span x-show="mode === 'dark'">
        Dark mode
    </span>
</div>
```

Collapsible sidebar

By default, the sidebar is only collapsible on mobile. You may make it collapsible on desktop as well.

You must [publish the configuration](#) in order to access this feature.

In `config/filament.php`, set the `layout.sidebar.is_collapsible_on_desktop` to `true`:

```
'layout' => [
  'sidebar' => [
    'is_collapsible_on_desktop' => true,
  ],
],
]
```

Non-sticky topbar

By default, the topbar sticks to the top of the page.

You may make the topbar scroll out of view instead by adding the following styles to your [theme](#) or by [registering a new stylesheet](#):

```
.filament-main-topbar {
  position: relative;
}
```

Building themes

Filament allows you to change the fonts and color scheme used in 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, and the Tailwind Typography plugin, [Autoprefixer](#), and [Tippy.js](#). You may install these through NPM:

```
npm install tailwindcss @tailwindcss/forms @tailwindcss/typography autoprefixer tippy.js --save-dev
```

To finish installing Tailwind, you must create a new `tailwind.config.js` file in the root of your project. The easiest way to do this is by running `npx tailwindcss init`.

In `tailwind.config.js`, register the plugins you installed, and add custom colors used by the form builder:

```

import colors from 'tailwindcss/colors' // [tl! focus:start]
import forms from '@tailwindcss/forms'
import typography from '@tailwindcss/typography' // [tl! focus:end]

export default {
    content: [
        './resources/**/*.blade.php',
        './vendor/filament/**/*.blade.php', // [tl! focus]
    ],
    darkMode: 'class',
    theme: {
        extend: {
            colors: { // [tl! focus:start]
                danger: colors.rose,
                primary: colors.blue,
                success: colors.green,
                warning: colors.yellow,
            }, // [tl! focus:end]
        },
    },
    plugins: [
        forms, // [tl! focus:start]
        typography, // [tl! focus:end]
    ],
}

```

You may specify your own colors, which will be used throughout the admin panel.

If you use Vite to compile assets, in your `vite.config.js` file, register the `filament.css` theme file:

```

import { defineConfig } from 'vite'
import laravel from 'laravel-vite-plugin'

export default defineConfig({
    plugins: [
        laravel({
            input: [
                // ...
                'resources/css/filament.css',
            ],
            // ...
        }),
    ],
})

```

And add Tailwind to the `postcss.config.js` file:

```

export default {
    plugins: {
        tailwindcss: {},
        autoprefixer: {},
    },
}

```

Or if you're using Laravel Mix instead of Vite, in your `webpack.mix.js` file, register Tailwind CSS as a PostCSS plugin:

```
const mix = require('laravel-mix')

mix.postCss('resources/css/filament.css', 'public/css', [
    require('tailwindcss'), // [tl! focus]
])
```

In `/resources/css/filament.css`, import Filament's vendor CSS:

```
@import '../../../../../vendor/filament/filament/resources/css/app.css';
```

Now, you may register the theme file in a service provider's `boot()` method:

```
use Filament\Facades\Filament;

Filament::serving(function () {
    // Using Vite
    Filament::registerViteTheme('resources/css/filament.css');

    // Using Laravel Mix
    Filament::registerTheme(
        mix('css/filament.css'),
    );
});
```

Loading Google Fonts

If you specify a custom font family in your `tailwind.config.js`, you may wish to import it via Google Fonts.

You must publish the configuration in order to access this feature.

Set the `google_fonts` config option to a new Google Fonts URL to load:

```
'google_fonts' => 'https://fonts.googleapis.com/css2?family=Inter:ital,wght@0,400;0,500;0,700;1,400;1,500;1,700&display=swap',
```

Changing the maximum content width

Filament exposes a configuration option that allows you to change the maximum content width of all pages.

You must publish the configuration in order to access this feature.

In `config/filament.php`, set the `layout.max_content_width` to any value between `xl` and `7xl`, or `full` for no max width:

```
'layout' => [
    'max_content_width' => 'full',
],
```

The default is `7xl`.

You may override the maximum content width for a specific page in the admin panel by using the `$maxContentWidth` property:

```
protected ?string $maxContentWidth = 'full';
```

Including frontend assets

You may register your own scripts and styles using the `registerScripts()` and `registerStyles()` methods in a service provider's `boot()` method:

```
use Filament\Facades\Filament;

Filament::registerScripts([
    asset('js/my-script.js'),
]);

Filament::registerStyles([
    'https://unpkg.com/tippy.js@6/dist/tippy.css',
    asset('css/my-styles.css'),
]);
```

You may pass `true` as a parameter to `registerScripts()` to load it before Filament's core JavaScript. This is useful for registering Alpine.js plugins from a CDN:

```
Filament::registerScripts([
    'https://cdn.jsdelivr.net/npm/@ryangjchandler/alpine-tooltip@0.x.x/dist/cdn.min.js',
], true);
```

Custom meta tags

You can add custom tags to the header, such as `<meta>` and `<link>`, using the following:

```
use Filament\Facades\Filament;
use Illuminate\Support\HtmlString;

Filament::pushMeta([
    new HtmlString('<link rel="manifest" href="/site.webmanifest" />'),
]);
```

Notification position

Filament allows you to customize the position of notifications.

In `config/filament.php`, set the `layout.notifications.alignment` to any value of `left`, `center` or `right` and `layout.notifications.vertical_alignment` to any value of `top`, `center` or `bottom`:

```
'layout' => [
    'notifications' => [
        'vertical_alignment' => 'top',
        'alignment' => 'center',
    ],
],
```

Render hooks

Filament allows you to render Blade content at various points in the admin panel layout. This is useful for integrations with packages like `wire-elements/modal` which require you to add a Livewire component to your app.

Here's an example, integrating `wire-elements/modal` with Filament in a service provider:

```
use Filament\Facades\Filament;
use Illuminate\Support\Facades\Blade;

Filament::registerRenderHook(
    'body.start',
    fn (): string => Blade::render('@livewire(\'livewire-ui-modal\')'),
);
```

You could also render view content from a file:

```
use Filament\Facades\Filament;
use Illuminate\Contracts\View\View;

Filament::registerRenderHook(
    'body.start',
    fn (): View => view('impersonation-banner'),
);
```

The available hooks are as follows:

- `body.start` - after `<body>`
- `body.end` - before `</body>`
- `head.start` - after `<head>`
- `head.end` - before `</head>`
- `content.start` - before page content
- `content.end` - after page content
- `footer.before` - before footer
- `footer.start` - start of footer content (centered)
- `footer.end` - end of footer content (centered)
- `footer.after` - after footer
- `sidebar.start` - before `sidebar` content
- `sidebar.end` - after `sidebar` content
- `scripts.start` - before scripts are defined
- `scripts.end` - after scripts are defined
- `styles.start` - before styles are defined
- `styles.end` - after styles are defined
- `global-search.start` - before `global search` input
- `global-search.end` - after `global search` input
- `user-menu.start` - before `user menu` input
- `user-menu.end` - after `user menu` input
- `user-menu.account.before` - before the account item in the `user menu`
- `user-menu.account.after` - after the account item in the `user menu`
- `page.header-widgets.start` - before page header widgets
- `page.header-widgets.end` - after page header widgets
- `page.footer-widgets.start` - before page footer widgets
- `page.footer-widgets.end` - after page footer widgets
- `page.actions.start` - before page actions

- `page.actions.end` - after page actions
- `resource.pages.list-records.table.start` - before the resource table
- `resource.pages.list-records.table.end` - after the resource table
- `resource.relation-manager.start` - before the relation manager table
- `resource.relation-manager.end` - after the relation manager table

Plugins

Plugins can be used to extend Filament's default behaviour and create reusable modules for use in multiple applications.

To create a new plugin, extend the `Filament\PluginServiceProvider` class provided by Filament:

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }

    // ...
}
```

The `PluginServiceProvider` extends the service provider from [Laravel Package Tools](#), so all `configurePackage()` options available there are available here as well.

Plugins must have a unique name property.

Registering plugins

Application plugins

If you're developing a plugin for a specific application, you should register the new service provider in your `config/app.php` file:

```
return [
    'providers' => [
        // ...
        \App\Providers\ExampleServiceProvider::class,
    ],
];
```

Laravel will load your service provider when bootstrapping and your plugin will be initialised.

Distributed plugins

Much like a normal Laravel package, you should add your service provider's fully qualified class name to the `extra.laravel.providers` array in your package's `composer.json` file:

```
{
    "extra": {
        "laravel": {
            "providers": [
                "Vendor\\Package\\ExampleServiceProvider"
            ]
        }
    }
}
```

This will ensure your service provider is automatically loaded by Laravel when the package is installed.

Resources

To register a custom resource, add the fully qualified class name to the `[$resources]` property in your service provider:

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;
use Vendor\Package\Resources\CustomResource;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $resources = [
        CustomResource::class,
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

Filament will automatically register your `Resource` and ensure that Livewire can discover it.

Pages

To register a custom page, add the fully qualified class name to the `[$pages]` property in your service provider:

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;
use Vendor\Package\Pages\CustomPage;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $pages = [
        CustomPage::class,
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

Filament will automatically register your `Page` and ensure that Livewire can discover it.

Widgets

To register a custom widget, add the fully qualified class name to the `$widgets` property in your service provider:

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;
use Vendor\Package\Widgets\CustomWidget;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $widgets = [
        CustomWidget::class,
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

Filament will automatically register your `Widget` and ensure that Livewire can discover it.

Frontend assets

Filament plugins can also register their own frontend assets. These assets will be included on all Filament related pages, allowing you to use your own CSS and JavaScript.

Stylesheets

To include a custom stylesheet, add it to the `$styles` property in your service provider. You should use a unique name as the key and the URL to the stylesheet as the value.

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $styles = [
        'my-package-styles' => __DIR__ . '/../dist/app.css',
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

Tailwind CSS

If you are using Tailwind classes, that are not used in Filament core, you need to compile your own Tailwind CSS file and bundle it with your plugin. Follow the Tailwind instructions for setup, but omit `@tailwind base` as this would overwrite the base styles if users customize their Filament theme.

After compilation, your Tailwind stylesheet may contain classes that are already used in Filament core. You should purge those classes with [awcodes/filament-plugin-purge](#) to keep the stylesheets size low.

Scripts

To include a custom script, add it to the `$scripts` property in your service provider. You should use a unique name as the key and the URL to the script as the value. These scripts will be added after the core Filament script.

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $scripts = [
        'my-package-scripts' => __DIR__ . '/../dist/app.js',
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

To add scripts before the core Filament script, use the `$beforeCoreScripts` property. This is useful if you want to hook into an Alpine event.

```
use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    protected array $beforeCoreScripts = [
        'my-package-scripts' => __DIR__ . '/../dist/app.js',
    ];

    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }
}
```

Providing data to the frontend

Whilst building your plugin, you might find the need to generate some data on the server and access it on the client.

To do this, use the `getScriptData()` method on your service provider and return an array of `string` keys and values that can be passed to converted into JSON:

```

use Filament\PluginServiceProvider;
use Illuminate\Support\Facades\Auth;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }

    protected function getScriptData(): array
    {
        return [
            'userId' => Auth::id(),
        ];
    }
}

```

You may now access this data in your scripts:

```

<script>
    console.log(window.filamentData.userId)
</script>

```

User menu

To register [user menu items](#) from your plugin, return them from the `getUserMenuItems()` method in your service provider:

```

use Filament\PluginServiceProvider;
use Spatie\LaravelPackageTools\Package;
use Vendor\Package\Pages\CustomPage;

class ExampleServiceProvider extends PluginServiceProvider
{
    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }

    protected function getUserMenuItems(): array
    {
        return [
            UserMenuItem::make()
                ->label('Settings')
                ->url(route('filament.pages.settings'))
                ->icon('heroicon-s-cog'),
        ];
    }
}

```

Filament will automatically register your `Page` and ensure that Livewire can discover it.

Commands, views, translations, migrations and more

Since the `PluginServiceProvider` extends the service provider from [Laravel Package Tools](#), you can use the `configurePackage` method to register [commands](#), [views](#), [translations](#), [migrations](#) and more.

ServingFilament Event

If you rely on data defined by Filament on `boot()` or through `Filament::serving()`, you can register listeners for the `Filament\Events\ServingFilament` event:

```
use Filament\Events\ServingFilament;
use Filament\PluginServiceProvider;
use Illuminate\Support\Facades\Event;
use Spatie\LaravelPackageTools\Package;

class ExampleServiceProvider extends PluginServiceProvider
{
    public function configurePackage(Package $package): void
    {
        $package->name('your-package-name');
    }

    public function packageConfiguring(Package $package): void
    {
        Event::listen(ServingFilament::class, [$this, 'registerStuff']);
    }

    protected function registerStuff(ServingFilament $event): void
    {
        // ...
    }
}
```

Testing

All examples in this guide will be written using [Pest](#). However, you can easily adapt this to a PHPUnit.

Since all pages in the admin panel 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 admin panel in your `TestCase`:

```
protected function setUp(): void
{
    parent::setUp();

    $this->actingAs(User::factory()->create());
}
```

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 `callPageAction()`:

```
use Filament\Pages\Actions\DeleteAction;
use function Pest\Livewire\livewire;

it('can delete', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->callPageAction(DeleteAction::class);

    $this->assertModelMissing($post);
});
```

You can ensure that a particular user is not able to see a `DeleteAction` using `assertPageActionHidden()`:

```
use Filament\Pages\Actions\DeleteAction;
use function Pest\Livewire\livewire;

it('can not delete', function () {
    $post = Post::factory()->create();

    livewire(PostResource\Pages>EditPost::class, [
        'record' => $post->getRouteKey(),
    ])
    ->assertPageActionHidden(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 `assertSet()` 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 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,
    ])
    ->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 function Pest\Livewire\livewire;

it('can list posts', function () {
    $category = Category::factory()
        ->has(Post::factory()->count(10))
        ->create();

    livewire(CategoryResource\RelationManagers\PostsRelationManager::class, [
        'ownerRecord' => $category,
    ])
    ->assertCanSeeTableRecords($category->posts);
});
```

Page actions

Calling actions

You can call a [page action](#) by passing its name or class to `callPageAction()`:

```
use function Pest\LiveWire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->callPageAction('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,
    ])
    ->callPageAction('send', data: [
        'email' => $email = fake()->email(),
    ])
    ->assertHasNoPageActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($email);
});
```

If you ever need to only set a page action's data without immediately calling it, you can use `setPageActionData()`:

```
use function Pest\Livewire\livewire;

it('can send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->mountPageAction('send')
    ->setPageActionData('send', data: [
        'email' => $email = fake()->email(),
    ])
});
});
```

Execution

To check if an action has been halted, you can use `assertPageActionHalted()`:

```
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,
    ])
    ->callPageAction('send')
    ->assertPageActionHalted('send');
});
});
```

Errors

`assertHasNoPageActionErrors()` 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 `assertHasPageActionErrors()`, 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,
    ])
    ->callPageAction('send', data: [
        'email' => Str::random(),
    ])
    ->assertHasPageActionErrors(['email' => ['email']]);
});
});
```

Pre-filled data

To check if a page action is pre-filled with data, you can use the `assertPageActionDataSet()` 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,
    ])
    ->mountPageAction('send')
    ->assertPageActionDataSet([
        'email' => $recipientEmail,
    ])
    ->callMountedPageAction()
    ->assertHasNoPageActionErrors();

    expect($invoice->refresh())
        ->isSent()->toBeTrue()
        ->recipient_email->toBe($recipientEmail);
});

});
```

Action State

To ensure that an action exists or doesn't in a table, you can use the `assertPageActionExists()` or `assertPageActionDoesNotExist()` method:

```
use function Pest\Livewire\livewire;

it('can send but not unsend invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertPageActionExists('send')
    ->assertPageActionDoesNotExist('unsend');
});
```

To ensure a page action is hidden or visible for a user, you can use the `assertPageActionHidden()` or `assertPageActionVisible()` methods:

```
use function Pest\Livewire\livewire;

it('can only print invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertPageActionHidden('send')
    ->assertPageActionVisible('print');
});
```

To ensure a page action is enabled or disabled for a user, you can use the `assertPageActionEnabled()` or `assertPageActionDisabled()` methods:

```
use function Pest\LiveWire\livewire;

it('can only print a sent invoice', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertPageActionDisabled('send')
        ->assertPageActionEnabled('print');
});
```

To ensure sets of actions exist in the correct order, you can use `assertPageActionsExistInOrder()`:

```
use function Pest\LiveWire\livewire;

it('can not send invoices', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertPageActionsExistInOrder(['send', 'export']);
});
```

Button appearance

To ensure an action has the correct label, you can use `assertPageActionHasLabel()` and `assertPageActionDoesNotHaveLabel()`:

```
use function Pest\LiveWire\livewire;

it('send action has correct label', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
        ->assertPageActionHasLabel('send', 'Email Invoice')
        ->assertPageActionDoesNotHaveLabel('send', 'Send');
});
```

To ensure an action's button is showing the correct icon, you can use `assertPageActionHasIcon()` or `assertPageActionDoesNotHaveIcon()`:

```
use function Pest\Livewire\livewire;

it('when enabled the send button has correct icon', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertPageActionEnabled('send')
    ->assertPageActionHasIcon('send', 'envelope-open')
    ->assertPageActionDoesNotHaveIcon('send', 'envelope');
});
```

To ensure an action's button is displaying the right color, you can use `assertPageActionHasColor()` or `assertPageActionDoesNotHaveColor()`:

```
use function Pest\Livewire\livewire;

it('actions display proper colors', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertPageActionHasColor('delete', 'danger')
    ->assertPageActionDoesNotHaveColor('print', 'danger');
});
```

URL

To ensure an action has the correct URL, you can use `assertPageActionHasUrl()`, `assertPageActionDoesNotHaveUrl()`, `assertPageActionShouldOpenUrlInNewTab()`, and `assertPageActionShouldNotOpenUrlInNewTab()`:

```
use function Pest\Livewire\livewire;

it('links to the correct Filament sites', function () {
    $invoice = Invoice::factory()->create();

    livewire(EditInvoice::class, [
        'invoice' => $invoice,
    ])
    ->assertPageActionHasUrl('filament', 'https://filamentphp.com/')
    ->assertPageActionDoesNotHaveUrl('filament', 'https://github.com/filamentphp/filament')
    ->assertPageActionShouldOpenUrlInNewTab('filament')
    ->assertPageActionShouldNotOpenUrlInNewTab('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!

High impact changes

Property and method changes to resource and page classes

- ▶ Changes to Resource classes
- ▶ Changes to List page classes
- ▶ Changes to Create page classes
- ▶ Changes to Edit page classes

Forms

The entire `Filament\Resources\Forms` namespace has been moved to `Filament\Forms`.

The `when()`, `only()` and `except()` methods have been removed. You may now pass a closure to any field configuration method, for example `hidden()`, which determines if the method should be applied. For more information, please see the [advanced forms documentation](#);

Fields

The `dependable()` method has been renamed to `reactive()`, to better describe its effects.

The `helpMessage()` method has been renamed to `helperText()`.

Checkbox

The `stacked()` method has been removed, and replaced with `inline(false)`.

Select

The `emptyOptionsMessage()` method has been renamed to `searchPrompt()`.

Tags input

The tags input component now writes to a JSON array by default. To continue using the old behavior, use `separator('')` method.

Toggle

The `stacked()` method has been removed, and replaced with `inline(false)`.

Layout components

Fieldset

The form components within the fieldset now need to be in their own `schema()` method, instead of being passed into `make()`.

Grid

The form components within the grid now need to be in their own `schema()` method, instead of being passed into `make()`.

Section

The form components within the section now need to be in their own `schema()` method, instead of being passed into `make()`.

Tabs

The `Filament\Resources\Forms\Tab` component has been moved to `Filament\Forms\Tabs\Tab`.

The form components within each tab now need to be in their own `schema()` method, instead of being passed into `make()`.

Tables

The entire `Filament\Resources\Tables` namespace has been moved to `Filament\Tables`.

The `only()` and `except()` methods have been removed. You may now pass a closure to any column or filter configuration method, which determines if the method should be applied.

Columns

Column class names now have `Column` at the end, for example `TextColumn` not `Text`.

The `currency()` method has been renamed to `money()`.

The `formatUsing()` method has been renamed to `formatStateUsing()`. It now accepts a `$state` parameter, instead of `$value`.

The `getValueUsing()` method has been renamed to `getStateUsing()`.

The `primary()` method has been removed from columns. All columns link to the record page by default unless another URL or action is specified for that column.

Filters

The filter class has been moved from `Filament\Resources\Tables\Filter` to `Filament\Tables\Filters\Filter`.

Filters now have a dedicated `query()` method for applying the query, instead of using the second parameter of the `make()` method. For more information, check out the [table builder filters documentation](#).

The `apply()` method of reusable filters must now have the following signature:

```
public function apply(Builder $query, array $data = []): Builder
{
    // ...
}
```

Published configuration updates

If you've published the v1.x `filament.php` configuration file, you should republish it:

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

If you had customized the `path`, `domain` or `default_filesystem_disk`, you should update the new file with these changes. If you're using `.env` variables for these settings, you won't need to make any changes when upgrading, and you may even choose to delete `filament.php`.

Users

Filament v2.x does not include a dedicated `filament_users` table as it did in v1.x. By default, all `App\Models\User`s can access the admin panel locally, and in production you must apply the `FilamentUser` interface to the model to control admin access. You can read more about this [here](#).

- ▶ **Recommended:** Are you using the `filament_users` table, but would like to switch to `App\Models\User`?
- ▶ Are you using the `filament_users` table, and would like to continue using it?

- Are you already using `App\Models\User`?

Medium impact changes

Relation managers

`HasMany` relation manager classes should now extend `Filament\Resources\RelationManagers\HasManyRelationManager`. `MorphMany` relation manager classes should now extend `Filament\Resources\RelationManagers\MorphManyRelationManager`. `BelongsToMany` relation manager classes should now extend `Filament\Resources\RelationManagers\BelongsToManyRelationManager`.

The `Filament\Resources\Forms\Form` class has been renamed to `Filament\Resources\Form`.

The `Filament\Resources\Tables\Table` class has been renamed to `Filament\Resources\Table`.

The following properties and method signatures been updated:

```
protected static ?string $inverseRelationship; // Protected the property. Added the `?string` type.

protected static ?string $recordTitleAttribute; // Renamed from `$primaryColumn`. Protected the property. Added the `?string` type.

protected static string $relationship; // Protected the property. Added the `string` type.
```

`Filament\Filament` facade renamed to `Filament\Facades\Filament`

You should be able to safely rename all instances of this class to the new one.

Roles

Filament now only uses policies for authorization, so you may implement whichever roles system you wish there. We recommend [spatie/laravel-permission](#).

You may remove any roles from the `App\Filament\Roles` directory, and delete any `authorization()` methods on your resources.

Low impact changes

`Filament::ignoreMigrations()` method removed

Since Filament doesn't have any migrations anymore, you don't need to ignore them.

Property changes to custom page classes

The following properties and method signatures been updated:

```
protected static ?string $title; // Protected the property. Added the `?string` type.

protected static ?string $navigationLabel; // Protected the property. Added the `?string` type.

protected static ?string $slug; // Protected the property. Added the `?string` type.
```

Theming changes

The theming system has entirely changed, to add support for Tailwind JIT's opacity features, which don't support static color codes.

Follow the instructions on the [appearance page](#) to find out how to compile your own Filament stylesheet.

Chapter 2

Form Builder

Installation

Requirements

Filament has a few requirements to run:

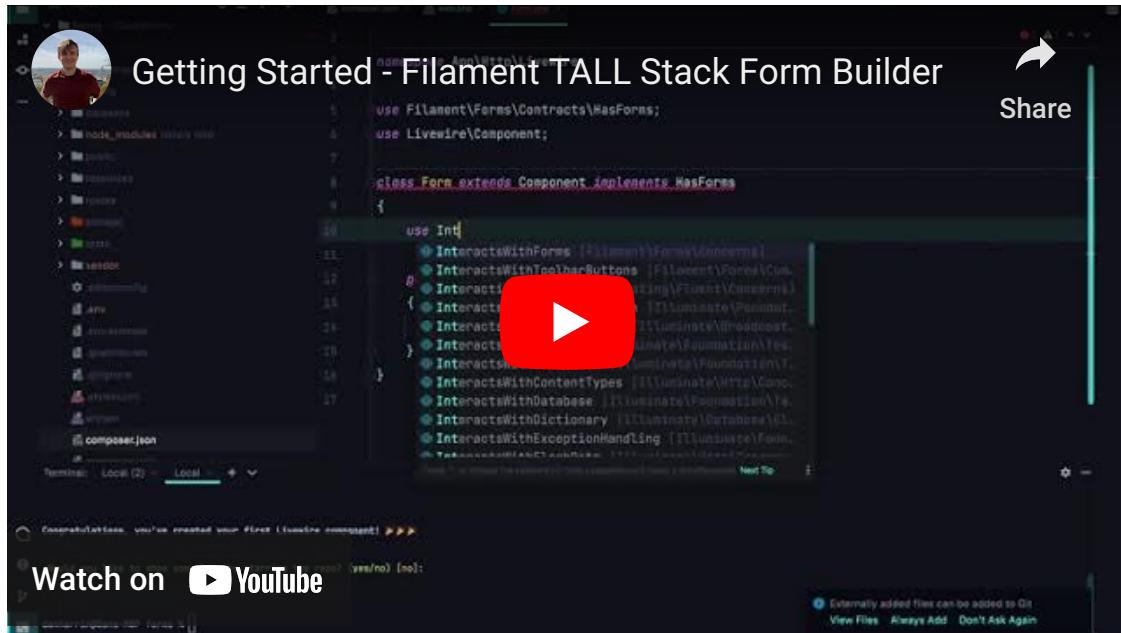
- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

The form builder comes pre-installed inside the [admin panel 2.x](#), but you must still follow the installation instructions below if you're using it in the rest of your app.

First, require the form builder using Composer:

```
composer require filament/forms:"^2.0"
```

New Laravel projects



To get started with the form builder quickly, you can set up [Livewire](#), [Alpine.js](#) and [Tailwind CSS](#) with these commands:

```
php artisan forms:install
npm install
npm run dev
```

These commands will ruthlessly overwrite existing files in your application, hence why we only recommend using this method for new projects.

You're now ready to start [building forms!](#)

Existing Laravel projects

The package uses the following dependencies:

- [Alpine.js](#)
- [PostCSS](#)
- [Tailwind CSS](#)
- [Tailwind CSS Forms plugin](#)
- [Tailwind CSS Typography plugin](#)

You may install these through NPM:

```
npm install alpinejs postcss tailwindcss @tailwindcss/forms @tailwindcss/typography --save-dev
```

Configuring Tailwind CSS

To finish installing Tailwind, you must create a new `tailwind.config.js` file in the root of your project. The easiest way to do this is by running `npx tailwindcss init`.

In `tailwind.config.js`, register the plugins you installed, and add custom colors used by the form builder:

```
import colors from 'tailwindcss/colors' // [tl! focus:start]
import forms from '@tailwindcss/forms'
import typography from '@tailwindcss/typography' // [tl! focus:end]

export default {
    content: [
        './resources/**/*.blade.php',
        './vendor/filament/**/*.blade.php', // [tl! focus]
    ],
    theme: {
        extend: {
            colors: { // [tl! focus:start]
                danger: colors.rose,
                primary: colors.blue,
                success: colors.green,
                warning: colors.yellow,
            }, // [tl! focus:end]
        },
    },
    plugins: [
        forms, // [tl! focus:start]
        typography, // [tl! focus:end]
    ],
}
```

Of course, you may specify your own custom `primary`, `success`, `warning` and `danger` colors, which will be used instead.

Bundling assets

New Laravel projects use Vite for bundling assets by default. However, your project may still use Laravel Mix. Read the steps below for the bundler used in your project.

Vite

If you're using Vite, you should manually install [Autoprefixer](#) through NPM:

```
npm install autoprefixer --save-dev
```

Create a `postcss.config.js` file in the root of your project, and register Tailwind CSS and Autoprefixer as plugins:

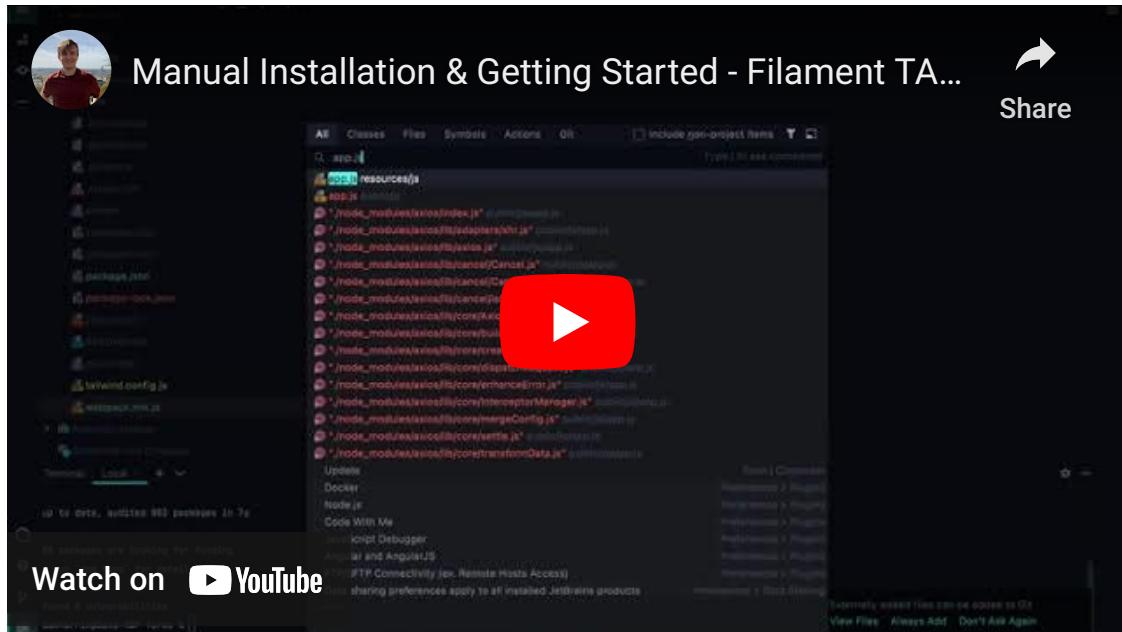
```
export default {
    plugins: {
        tailwindcss: {},
        autoprefixer: {}
    }
}
```

You may also want to update your `vite.config.js` file to refresh the page after Livewire components or custom form components have been updated:

```
import { defineConfig } from 'vite'
import laravel, { refreshPaths } from 'laravel-vite-plugin' // [tl! focus]

export default defineConfig({
    plugins: [
        laravel({
            input: [
                'resources/css/app.css',
                'resources/js/app.js',
            ],
            refresh: [ // [tl! focus:start]
                ...refreshPaths,
                'app/Http/Livewire/**',
                'app/Forms/Components/**',
            ], // [tl! focus:end]
        }),
    ],
})
```

Laravel Mix



In your `webpack.mix.js` file, register Tailwind CSS as a PostCSS plugin:

```
const mix = require('laravel-mix')

mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css', [
        require('tailwindcss'), // [tl! focus]
    ])
])
```

Configuring styles

In `/resources/css/app.css`, import `filament/forms` vendor CSS and [Tailwind CSS](#):

```
@import '../../../../../vendor/filament/forms/dist/module.esm.css';

@tailwind base;
@tailwind components;
@tailwind utilities;
```

Configuring scripts

In `/resources/js/app.js`, import [Alpine.js](#), the `filament/forms` and `filament/notifications` plugins, and register them:

```
import Alpine from 'alpinejs'
import FormsAlpinePlugin from '../../../../../vendor/filament/forms/dist/module.esm'
import NotificationsAlpinePlugin from '../../../../../vendor/filament/notifications/dist/module.esm'

Alpine.plugin(FormsAlpinePlugin)
Alpine.plugin(NotificationsAlpinePlugin)

window.Alpine = Alpine

Alpine.start()
```

Compiling assets

Compile your new CSS and JS assets using `npm run dev`.

Configuring layout

Finally, create a new `resources/views/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>
        @vite(['resources/css/app.css', 'resources/js/app.js'])
        @livewireStyles
        @livewireScripts
        @stack('scripts')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @livewire('notifications')
    </body>
</html>

```

You're now ready to start [building forms!](#)

Publishing configuration

If you wish, you may publish the configuration of the package using:

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

Publishing translations

If you wish to translate the package, you may publish the language files using:

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

Since this package depends on other Filament packages, you may wish to translate those as well:

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

Upgrading

To upgrade the package to the latest version, you must run:

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

We recommend adding the `filament:upgrade` command to your `composer.json`'s `post-update-cmd` to run it automatically:

```
"post-update-cmd": [  
    // ...  
    "@php artisan filament:upgrade"  
,
```

Getting Started



Preparing your Livewire component

Implement the `HasForms` interface and use the `InteractsWithForms` trait:

```
<?php

namespace App\Http\Livewire;

use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms // [tl! focus]
{
    use Forms\Concerns\InteractsWithForms; // [tl! focus]

    public function render(): View
    {
        return view('edit-post');
    }
}
```

In your Livewire component's view, render the form:

```
<form wire:submit.prevent="submit">
{{ $this->form }}

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

Finally, add any fields and layout components to the Livewire component's `getFormSchema()` method:

```
<?php

namespace App\Http\Livewire;

use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $title;
    public $content;

    public function mount(): void
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    }

    protected function getFormSchema(): array // [tl! focus:start]
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
            // ...
        ];
    } // [tl! focus:end]

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

    public function render(): View
    {
        return view('edit-post');
    }
}
```

Visit your Livewire component in the browser, and you should see the form components from `getFormSchema()`.

Title *

Content



Edit Preview

Initializing forms

You must initialize forms when the Livewire component is first loaded. This is done with the `fill()` form method, often called in the `mount()` method of the Livewire component.

For your fields to hold data, they should have a corresponding property on your Livewire component, just as in Livewire normally.

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class CreatePost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public $title = '';
    public $content = '';

    public function mount(): void // [tl! focus:start]
    {
        $this->form->fill();
    } // [tl! focus:end]

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')
                ->default('Status Update') // [tl! focus]
                ->required(),
            Forms\Components\MarkdownEditor::make('content'),
        ];
    }

    public function render(): View
    {
        return view('create-post');
    }
}

```

You may customize what happens after fields are filled using the `afterStateHydrated()` method.

Filling forms with data

To fill a form with data, call the `fill()` method on your form, and pass an array of data to fill it with:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $title;
    public $content;

    public function mount(): void // [tl! focus:start]
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    } // [tl! focus:end]

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
        ];
    }

    public function render(): View
    {
        return view('edit-post');
    }
}

```

Getting data from forms

To get all form data in an array, call the `getState()` method on your form.

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class CreatePost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public $title = '';
    public $content = '';

    public function mount(): void
    {
        $this->form->fill();
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
        ];
    }

    public function create(): void // [tl! focus:start]
    {
        Post::create($this->form->getState());
    } // [tl! focus:end]

    public function render(): View
    {
        return view('create-post');
    }
}

```

When `getState()` is run:

1. Validation rules are checked, and if errors are present, the form is not submitted.
2. Any pending file uploads are stored permanently in the filesystem.
3. Field relationships, if they are defined, are saved.

You may transform the value that is dehydrated from a field using the `dehydrateStateUsing()` method.

Registering a model

You may register a model to a form. The form builder is able to use this model to unlock DX features, such as:

- Automatically retrieving the database table name when using database validation rules like `exists` and `unique`.
- Automatically attaching relationships to the model when the form is saved, when using fields such as the `Select`, `Repeater`, `SpatieMediaLibraryFileUpload`, or `SpatieTagsInput`.

Pass a model instance to a form using the `getFormModel()` method:

```
<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $title;
    public $content;
    public $tags;

    public function mount(): void
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\SpatieTagsInput::make('tags'),
        ];
    }

    protected function getFormModel(): Post // [tl! focus:start]
    {
        return $this->post;
    } // [tl! focus:end]

    public function render(): View
    {
        return view('edit-post');
    }
}
```

Alternatively, you may pass the model instance to the field that requires it directly, using the `model()` method:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $title;
    public $content;
    public $tags;

    public function mount(): void
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\SpatieTagsInput::make('tags')->model($this->post), // [tl! focus]
        ];
    }

    public function render(): View
    {
        return view('edit-post');
    }
}

```

You may now use [field relationships](#);

Registering a model class

In some cases, the model instance is not available until the form has been submitted. For example, in a form that creates a post, the post model instance cannot be passed to the form before it has been submitted.

You may receive some of the same benefits of registering a model by registering its class instead:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class CreatePost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public $title = '';
    public $content = '';
    public $categories = [];

    public function mount(): void
    {
        $this->form->fill();
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')
                ->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\Select::make('categories')
                ->multiple()
                ->relationship('categories', 'name'),
        ];
    }

    protected function getFormModel(): string // [tl! focus:start]
    {
        return Post::class;
    } // [tl! focus:end]

    public function render(): View
    {
        return view('create-post');
    }
}

```

You may now use [field relationships](#).

Field relationships

Some fields, such as the `Select`, `Repeater`, `SpatieMediaLibraryFileUpload`, or `SpatieTagsInput` are able to interact with model relationships.

For example, `Select` can be used to attach multiple records to a `BelongsToMany` relationship. When [registering a model](#) to the form or component, these relationships will be automatically saved to the pivot table [when `getState\(\)` is called](#):

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $title;
    public $content;
    public $categories;

    public function mount(): void
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')
                ->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\Select::make('categories')
                ->multiple()
                ->relationship('categories', 'name'),
        ];
    }

    protected function getFormModel(): Post // [tl! focus:start]
    {
        return $this->post;
    }

    public function save(): void
    {
        $this->post->update(
            $this->form->getState(),
        );
    } // [tl! focus:end]

    public function render(): View
    {
        return view('edit-post');
    }
}

```

Saving field relationships manually

In some cases, the model instance is not available until the form has been submitted. For example, in a form that creates a post, the post model instance cannot be passed to the form before it has been submitted. In this case, you will pass the model class instead, but any field relationships will need to be saved manually after.

In this situation, you may call the `model()` and `saveRelationships()` methods on the form after the instance has been created:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Model;
use Livewire\Component;

class CreatePost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public $title = '';
    public $content = '';
    public $tags = [];

    public function mount(): void
    {
        $this->form->fill();
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\SpatieTagsInput::make('tags'),
        ];
    }

    protected function getFormModel(): string
    {
        return Post::class;
    }

    public function create(): void
    {
        $post = Post::create($this->form->getState());

        $this->form->model($post)->saveRelationships(); // [tl! focus]
    }

    public function render(): View
    {
        return view('create-post');
    }
}

```

Saving relationships when the field is hidden

By default, relationships will only be saved if the field is visible. For example, if you have a `Repeater` field that is only visible on a certain condition, the relationships will not be saved when it is hidden.

This might cause unexpected behaviour if you still want to save the relationship, even when the field is hidden. To force relationships to be saved, you may call the `saveRelationshipsWhenHidden()` method on the form component:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')
    ->visible(fn (Closure $get): bool => $get('has_attachments'))
    ->saveRelationshipsWhenHidden();
```

Using multiple forms

By default, the `InteractsWithForms` trait only handles one form per Livewire component. To change this, you can override the `getForms()` method to return more than one form, each with a unique name:

```

<?php

namespace App\Http\Livewire;

use App\Models\Author;
use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Author $author;
    public Post $post;

    public $title;
    public $content;

    public $name;
    public $email;

    public function mount(): void
    {
        $this->postForm->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);

        $this->authorForm->fill([
            'name' => $this->author->name,
            'email' => $this->author->email,
        ]);
    }

    protected function getPostFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
        ];
    }

    protected function getAuthorFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('name')->required(),
            Forms\Components\TextInput::make('email')->email()->required(),
        ];
    }

    public function savePost(): void
    {
        $this->post->update(
            $this->postForm->getState(),
    }
}

```

```
        );
    }

    public function saveAuthor(): void
    {
        $this->author->update(
            $this->authorForm->getState(),
        );
    }

    protected function getForms(): array // [tl! focus:start]
    {
        return [
            'postForm' => $this->makeForm()
                ->schema($this->getPostFormSchema())
                ->model($this->post),
            'authorForm' => $this->makeForm()
                ->schema($this->getAuthorFormSchema())
                ->model($this->author),
        ];
    } // [tl! focus:end]

    public function render(): View
    {
        return view('edit-post');
    }
}
```

Scoping form data to an array property

You may scope the entire form data to a single array property on your Livewire component. This will allow you to avoid having to define a new property for each field:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Forms;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class EditPost extends Component implements Forms\Contracts\HasForms
{
    use Forms\Concerns\InteractsWithForms;

    public Post $post;

    public $data; // [tl! focus]

    public function mount(): void
    {
        $this->form->fill([
            'title' => $this->post->title,
            'content' => $this->post->content,
        ]);
    }

    protected function getFormSchema(): array
    {
        return [
            Forms\Components\TextInput::make('title')->required(),
            Forms\Components\MarkdownEditor::make('content'),
            Forms\Components\SpatieTagsInput::make('tags'),
        ];
    }

    protected function getFormModel(): Post
    {
        return $this->post;
    }

    protected function getFormStatePath(): string // [tl! focus:start]
    {
        return 'data';
    } // [tl! focus:end]

    public function render(): View
    {
        return view('edit-post');
    }
}

```

In this example, all data from your form will be stored in the `$data` array.

Fields

Getting started

Field classes can be found in the `Filament\Form\Components` namespace.

Fields reside within the schema of your form, alongside any [layout components](#).

If you're using the fields in a Livewire component, you can put them in the `getFormSchema()` method:

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

If you're using them in admin panel resources or relation managers, you must put them in the `$form->schema()` method:

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

Fields may be created using the static `make()` method, passing its name. The name of the field should correspond to a property on your Livewire component. You may use [Livewire's "dot syntax"](#) to bind fields to nested properties such as arrays and Eloquent models.

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
```

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 by using 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 inside the admin panel this only works on Create Pages, as Edit Pages will always fill the data from the model.

Helper messages and hints

Sometimes, you may wish to provide extra information for the user of the form. For this purpose, you may use helper messages and hints.

Help messages are displayed below the field. The `helperText()` method supports Markdown formatting:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->helperText('Your full name here, including any middle names.')
```

Hints can be used to display text adjacent to its label:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')->hint('[Forgotten your password?] (forgotten-password)')
```

Hints may also have an icon rendered next to them:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->hint('Translatable')
    ->hintIcon('heroicon-s-translate')
```

Hints may have a `color()`. By default it's gray, but you may use `primary`, `success`, `warning`, or `danger`:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->hint('Translatable')
    ->hintColor('primary')
```

Custom attributes

The HTML attributes of the field's wrapper can be customized by passing an array of `extraAttributes()`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->extraAttributes(['title' => 'Text input'])
```

To add additional HTML attributes to the input itself, use `extraInputAttributes()`:

```
use Filament\Forms\Components>Select;

Select::make('categories')
->extraInputAttributes(['multiple' => true])
```

Disabling

You may disable a field to prevent it from being edited:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->disabled()
```

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())
```

Please note that disabling a field does not prevent it from being saved, and a skillful user could manipulate the HTML of the page and alter its value.

To prevent a field from being saved, use the `dehydrated(false)` method:

```
Toggle::make('is_admin')->dehydrated(false)
```

Alternatively, you may only want to save a field conditionally, maybe if the user is an admin:

```
Toggle::make('is_admin')
->disabled(! auth()->user()->isAdmin())
->dehydrated(auth()->user()->isAdmin())
```

If you're using the [admin panel](#) and only want to save disabled fields on the [Create page of a resource](#):

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

TextInput::make('slug')
->disabled()
->dehydrated(fn (Page $livewire) => $livewire instanceof CreateRecord)
```

Hiding

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

Most fields will be autofocusable. Typically, you should aim for the first significant field in your form to be autofocus for the best user experience.

```
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. You may customize this using the `placeholder()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->placeholder('John Doe')
```

Global settings

If you wish to change the default behaviour of a field 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 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 on each field individually:

```
use Filament\Forms\Components\Checkbox;

Checkbox::make('is_admin')->inline()
```

Text input

The text input allows you to interact with a string:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
```

Name

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()
    ->numeric()
    ->password()
    ->tel()
    ->url()
```

You may instead use the `type()` method to pass another [HTML input type](#):

```
use Filament\Forms\Components\TextInput;

TextInput::make('backgroundColor')->type('color')
```

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 may 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)
```

In addition, you may validate the minimum and maximum value of the input by setting the `minValue()` and `maxValue()` methods:

```
use Filament\Forms\Components\TextInput;

TextInput::make('number')
->numeric()
->minValue(1)
->maxValue(100)
```

You may set the autocomplete configuration for the text field using the `autocomplete()` method:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
->password()
->autocomplete('new-password')
```

As a shortcut for `autocomplete="off"`, you may `disableAutocomplete()`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('password')
->password()
->disableAutocomplete()
```

For more complex autocomplete options, text inputs also support [datalists](#).

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 {
    $component->telRegex('/^[\+]*[(]{0,1}[0-9]{1,4}[)]{0,1}[-\s\.\/0-9]*$/');
});
```

Affixes

You may place text before and after the input using the `prefix()` and `suffix()` methods:

```
use Filament\Forms\Components\TextInput;

TextInput::make('domain')
->url()
->prefix('https://')
->suffix('.com')
```

Domain

<https://> .com

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()
->prefixIcon('heroicon-s-external-link')
->suffixIcon('heroicon-s-external-link')
```

You may render an action before and after the input using the `prefixAction()` and `suffixAction()` methods:

```
use Filament\Forms\Components\Actions\Action;
use Filament\Forms\Components\TextInput;

TextInput::make('domain')
->suffixAction(fn (?string $state): Action =>
    Action::make('visit')
        ->icon('heroicon-s-external-link')
        ->url(
            filled($state) ? "https://{$state}" : null,
            shouldOpenInNewTab: true,
        ),
)
```

Input masking

Input masking is the practice of defining a format that the input value must conform to.

In Filament, you may interact with the `Mask` object in the `mask()` method to configure your mask:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')
->mask(fn (TextInput\Mask $mask) => $mask->pattern('+{7} (000) 000-00-00'))
```

Under the hood, masking is powered by [imaskjs](#). The vast majority of its masking features are also available in Filament. Reading their [guide](#) first, and then approaching the same task using Filament is probably the easiest option.

You may define and configure a numeric mask to deal with numbers:

```
use Filament\Forms\Components\TextInput;

TextInput::make('number')
->numeric()
->mask(fn (TextInput\Mask $mask) => $mask
    ->numeric()
    ->decimalPlaces(2) // Set the number of digits after the decimal point.
    ->decimalSeparator(',') // Add a separator for decimal numbers.
    ->integer() // Disallow decimal numbers.
    ->mapToDecimalSeparator(['.']) // Map additional characters to the decimal separator.
    ->minValue(1) // Set the minimum value that the number can be.
    ->maxValue(100) // Set the maximum value that the number can be.
    ->normalizeZeros() // Append or remove zeros at the end of the number.
    ->padFractionalZeros() // Pad zeros at the end of the number to always maintain the
maximum number of decimal places.
->thousandsSeparator(',',) // Add a separator for thousands.
)
```

Enum masks limit the options that the user can input:

```
use Filament\Forms\Components\TextInput;

TextInput::make('code')->mask(fn (TextInput\Mask $mask) => $mask->enum(['F1', 'G2', 'H3']))
```

Range masks can be used to restrict input to a number range:

```
use Filament\Forms\Components\TextInput;

TextInput::make('code')->mask(fn (TextInput\Mask $mask) => $mask
->range()
->from(1) // Set the lower limit.
->to(100) // Set the upper limit.
->maxLength(100), // Pad zeros at the start of smaller numbers.
)
```

In addition to simple patterns, you may also define multiple pattern blocks:

```
use Filament\Forms\Components\TextInput;

TextInput::make('cost')->mask(fn (TextInput\Mask $mask) => $mask
->patternBlocks([
    'money' => fn (Mask $mask) => $mask
        ->numeric()
        ->thousandsSeparator(',')
        ->decimalSeparator('.'),
])
->pattern('$money'),
)
```

There is also a `money()` method that is able to define easier formatting for currency inputs. This example, the symbol prefix is `€`, there is a `,` thousands separator, and two decimal places:

```
use Filament\Forms\Components\TextInput;

TextInput::make('cost')->mask(fn (TextInput\Mask $mask) => $mask->money(prefix: '€',
thousandsSeparator: ',', decimalPlaces: 2))
```

You can also control whether the number is signed or not. While the default is to allow both negative and positive numbers, `isSigned: false` allows only positive numbers:

```
use Filament\Forms\Components\TextInput;

TextInput::make('cost')->mask(fn (TextInput\Mask $mask) => $mask->money(prefix: '€',
thousandsSeparator: ',', decimalPlaces: 2, isSigned: false))
```

Datalists

You may specify datalist options for a text input using the `datalist()` method:

```
TextInput::make('manufacturer')
->datalist([
    'BWM',
    'Ford',
    'Mercedes-Benz',
    'Porsche',
    'Toyota',
    'Tesla',
    'Volkswagen',
])
```

Manufacturer

BWM
Ford
Mercedes-Benz
Porsche
Toyota
Tesla

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 for strictly predefined options, check out [select fields](#).

Select

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',
])
])
```

In the `options()` array, the array keys are saved, and the array values will be the label of each option in the dropdown.

Status

Select an option 

You may enable a search input to allow easier access to many options, using the `searchable()` method:

```
use Filament\Forms\Components>Select;

Select::make('authorId')
->label('Author')
->options(User::all()->pluck('name', 'id'))
->searchable()
```

Author

Taylor Otwell

Mohamed Said

Dries Vints

James Brooks

Nuno Maduro

Mior Muhammad Zaki Mior Khairuddin

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 `getOptionLabelUsing()` method accepts a callback that transforms the selected option `$value` into a label.

```
Select::make('authorId')
->searchable()
->getSearchResultsUsing(fn (string $search) => User::where('name', 'like', "%{$search}%")-
>limit(50)->pluck('name', 'id'))
->getOptionLabelUsing(fn ($value): ?string => User::find($value)?->name),
```

You can prevent the placeholder from being selected using the `disablePlaceholderSelection()` method:

```
use Filament\Forms\Components\Select;

Select::make('status')
->options([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
->default('draft')
->disablePlaceholderSelection()
```

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',
])
```

Technologies

Select an option

Alpine.js × Laravel ×

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;

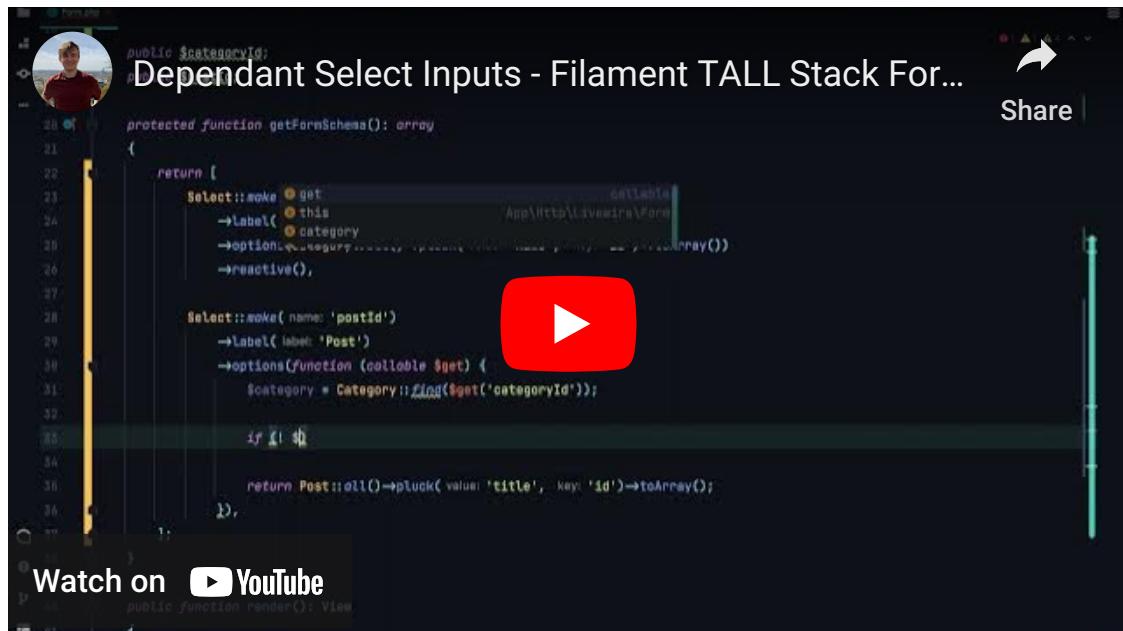
class App extends Model
{
    protected $casts = [
        'technologies' => 'array',
    ];

    // ...
}
```

Instead of `getOptionLabelUsing()`, the `getOptionLabelsUsing()` method can be used to transform the selected options' `$value`s into labels.

Dependant selects

Commonly, you may desire "dependant" select inputs, which populate their options based on the state of another.



Some of the techniques described in the [advanced forms](#) section are required to create dependant selects. These techniques can be applied across all form components for many dynamic customization possibilities.

Populating automatically from a relationship

You may employ the `relationship()` method of the `Select` to configure a `BelongsTo` relationship to automatically retrieve and save options from:

```
use Filament\Forms\Components\Select;

Select::make('authorId')
    ->relationship('author', 'name')
```

The `multiple()` method may be used in combination with `relationship()` to automatically populate from a `BelongsToMany` relationship:

```
use Filament\Forms\Components>Select;

Select::make('technologies')
    ->multiple()
    ->relationship('technologies', 'name')
```

To set this functionality up, **you must also follow the instructions set out in the [field relationships](#) section.** If you're using the [admin panel](#), you can skip this step.

If you'd like to populate the 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('authorId')
    ->relationship('author', 'name')
    ->preload()
```

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('authorId')
    ->relationship('author', 'name', fn (Builder $query) => $query->withTrashed())
```

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 should 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('authorId')
    ->relationship('author', 'full_name')
```

Alternatively, you can use the `getOptionLabelFromRecordUsing()` method to transform the selected option's Eloquent model into a label. But please note, this is much less performant than using a virtual column:

```
use Filament\Forms\Components>Select;
use Illuminate\Database\Eloquent\Model;

Select::make('authorId')
    ->relationship('author', 'first_name')
    ->getOptionLabelFromRecordUsing(fn (Model $record) => "{$record->first_name} {$record->last_name}")
```

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 a fieldset. 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)->titleColumnName('name'),
    MorphToSelect\Type::make(Post::class)->titleColumnName('title'),
])
```

The `titleColumnName()` is used to extract the titles out of each product or post. You can choose to extract the option labels using `getOptionLabelFromRecordUsing` instead if you wish:

```
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)->titleColumnName('title'),
])
```

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)
        ->titleColumnName('name')
        ->modifyOptionsQueryUsing(fn (Builder $query) => $query->whereBelongsTo($this->team)),
    MorphToSelect\Type::make(Post::class)
        ->titleColumnName('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()`, `allowHtml()`, and `optionsLimit()`.

Creating new records

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;
use Illuminate\Database\Eloquent\Model;

Select::make('authorId')
    ->relationship('author', 'name')
    ->createOptionForm([
        Forms\Components\TextInput::make('name')
            ->required(),
        Forms\Components\TextInput::make('email')
            ->required()
            ->email(),
    ]),
]
```

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.

Since HTML does not support nested `<form>` elements, you must also render the modal outside the `<form>` in the view. If you're using the [admin panel](#), this is included already:

```
<form wire:submit.prevent="submit">
    {{ $this->form }}

    <button type="submit">
        Submit
    </button>
</form>

{{ $this->modal }}
```

Checkbox

The checkbox component, similar to a [toggle](#), allows you to interact a boolean value.

```
use Filament\Forms\Components\Checkbox;

Checkbox::make('is_admin')
```



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



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',
    ];

    // ...
}
```

Toggle

The toggle component, similar to a [checkbox](#), allows you to interact a boolean value.

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
```



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()
```

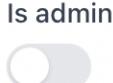


Is admin

When the toggle is stacked, its label is above it:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')->inline(false)
```



Toggles may also use an "on icon" and an "off icon". These are displayed on its handle and could provide a greater indication to what your field represents. The parameter to each method must contain the name of a Blade icon component:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
    ->onIcon('heroicon-s-lightning-bolt')
    ->offIcon('heroicon-s-user')
```

You may also customize the color representing each state. These may be either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Forms\Components\Toggle;

Toggle::make('is_admin')
    ->onColor('success')
    ->offColor('danger')
```



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',
    ];

    // ...
}
```

Checkbox list

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',
    ])
)
```

Technologies

- TailwindCSS
- 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',
    ];

    // ...
}
```

You may organize options into columns by using the `columns()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
->options([
    'tailwind' => 'Tailwind CSS',
    'alpine' => 'Alpine.js',
    'laravel' => 'Laravel',
    'livewire' => 'Laravel Livewire',
])
->columns(2)
```

Technologies

TailwindCSS
 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.

You may also allow users to toggle all checkboxes at once using the `bulkToggleable()` method:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
->options([
    'tailwind' => 'Tailwind CSS',
    'alpine' => 'Alpine.js',
    'laravel' => 'Laravel',
    'livewire' => 'Laravel Livewire',
])
->bulkToggleable()
```

Populating automatically from a relationship

You may employ the `relationship()` method to configure a relationship to automatically retrieve and save options from:

```
use Filament\Forms\Components\CheckboxList;

CheckboxList::make('technologies')
->relationship('technologies', 'name')
```

To set this functionality up, **you must also follow the instructions set out in the [field relationships](#) section**. If you're using the [admin panel](#), you can skip this step.

You may customize the database query that retrieves options using the third parameter of the `relationship()` method:

```
use Filament\Forms\Components\CheckboxList;
use Illuminate\Database\Eloquent\Builder;

CheckboxList::make('technologies')
    ->relationship('technologies', 'name', fn (Builder $query) => $query->withTrashed())
```

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 should 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('participants')
    ->relationship('participants', 'full_name')
```

Alternatively, you can use the `getOptionLabelFromRecordUsing()` method to transform the selected option's Eloquent model into a label. But please note, this is much less performant than using a virtual column:

```
use Filament\Forms\Components\CheckboxList;
use Illuminate\Database\Eloquent\Model;

CheckboxList::make('participants')
    ->relationship('participants', 'first_name')
    ->getOptionLabelFromRecordUsing(fn (Model $record) => "{$record->first_name} {$record->last_name}")
```

Radio

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

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 options array so the right description matches the right option.

If you want a simple boolean radio button group, with "Yes" and "No" options, you can use the `boolean()` method:

```
Radio::make('feedback')
->label('Do you like this post?')
->boolean()
```

Do you like this post?

- Yes**
- No**

You may wish to display the options `inline()` with the label:

```
Radio::make('feedback')
->label('Do you like this post?')
->boolean()
->inline()
```

Do you like this post? Yes No

Date-time picker

The date-time picker provides an interactive interface for selecting a date and 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

Dec 10, 2021 15:30:00



Date of birth

Sep 24, 1973



Alarm at

08:00:00



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')
->minDate(now() ->subYears(150))
->maxDate(now())
```

Date of birth

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')
```

You may also 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')->displayFormat('d/m/Y')
```

Date of birth

When using the time picker, you may disable the seconds input using the `withoutSeconds()` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')->withoutSeconds()
```

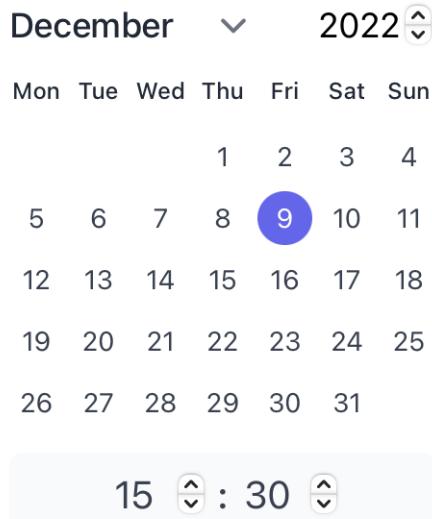
You may also customize the input interval for increasing the hours / minutes / seconds using the `hoursStep()`, `minutesStep()` or `secondsStep()` methods:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')
    ->hoursStep(2)
    ->minutesStep(15)
    ->secondsStep(10)
```

Published at

Dec 9, 2022 15:30



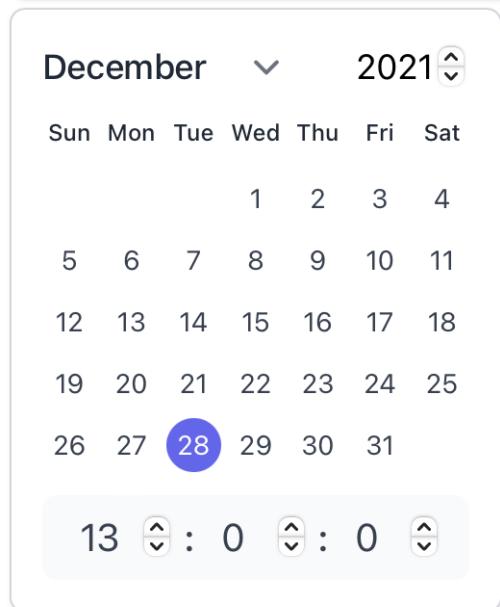
In some countries, the first day of the week is not Monday. To customize the first day of the week in the date picker, use the `forms.components.date_time_picker.first_day_of_week` config option, or the `firstDayOfWeek()` method on the component. 0 to 7 are accepted values, with Monday as 1 and Sunday as 7 or 0:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('published_at')->firstDayOfWeek(7)
```

Published at

Dec 28, 2021 13:00:00



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')->weekStartsOnMonday()
DateTimePicker::make('published_at')->weekStartsOnSunday()
```

To disable specific dates:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('date')
->label('Appointment date')
->minDate(now())
->maxDate(Carbon::now()->addDays(30))
->disabledDates(['2022-10-02', '2022-10-05', '2022-10-15'])
```

You may change the calendar icon using the `icon()` method:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('date')
->icon('heroicon-o-calendar')
```

Alternatively, you may remove the icon altogether by passing `false`:

```
use Filament\Forms\Components\DateTimePicker;

DateTimePicker::make('date')
->icon(false)
```

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.

File upload

The file upload field is based on [Filepond](#).

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
```

Attachment

Drag & Drop your files or [Browse](#)

By default, files will be uploaded publicly to your default storage disk.

Please note, 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 that files are saved in, and their visibility, use the `disk()`, `directory()` and `visibility` methods:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->disk('s3')
->directory('form-attachments')
->visibility('private')
```

Please note, 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](#).

By default, a random file name will be generated for newly-uploaded files. To instead preserve the original filenames of the uploaded files, use the `preserveFilenames()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')->preserveFilenames()
```

You may completely customize how file names are generated using the `getUploadedFileNameForStorageUsing()` method, and returning a string from the callback:

```
use Livewire\TemporaryUploadedFile;

FileUpload::make('attachment')
->getUploadedFileNameForStorageUsing(function (TemporaryUploadedFile $file): string {
    return (string) str($file->getClientOriginalName())->prepend('custom-prefix-');
})
```

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 name/s of your uploaded files.

You may restrict the types of files that may be uploaded using the `acceptedFileTypes()` method, and passing an array of MIME types. You may also use the `image()` method as shorthand to allow all image MIME types.

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('document')->acceptedFileTypes(['application/pdf'])
FileUpload::make('image')->image()
```

You may also restrict the size of uploaded files, in kilobytes:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachment')
->minSize(512)
->maxSize(1024)
```

To customize Livewire's default file upload validation rules, including the 12MB file size maximum, please refer to its [documentation](#).

Filepond allows you to crop and resize images before they are uploaded. You can customize this behaviour using the `imageResizeMode()`, `imageCropAspectRatio()`, `imageResizeTargetHeight()` and

`imageResizeTargetWidth()` methods. `imageResizeMode()` should be set for the other methods to have an effect - either `force`, `cover`, or `contain`.

```
use Filament\Forms\Components\FileUpload;

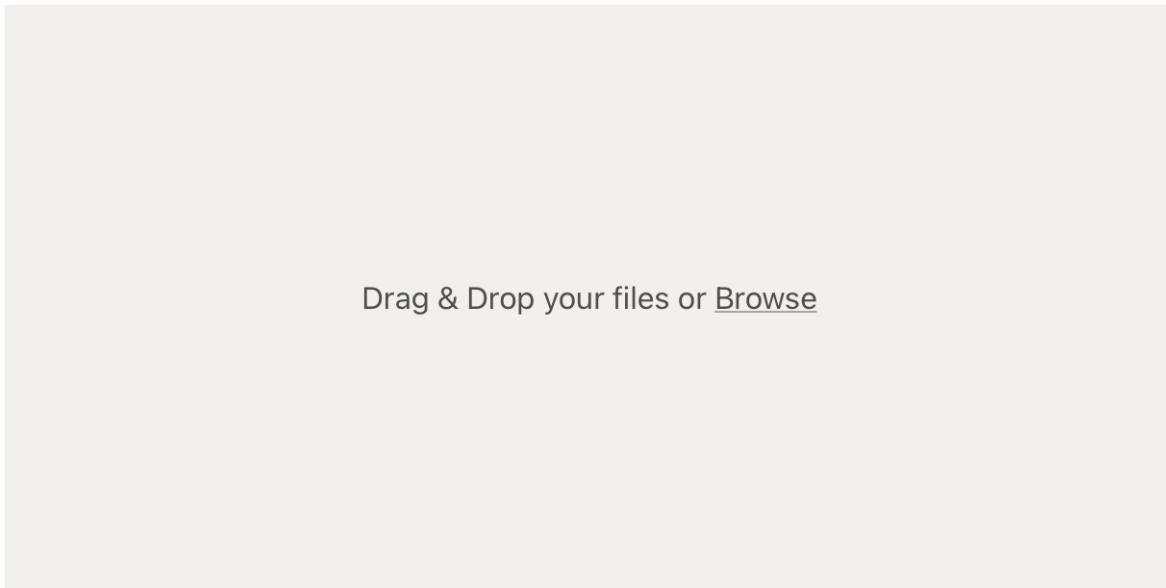
FileUpload::make('image')
    ->image()
    ->imageSizeMode('cover')
    ->imageCropAspectRatio('16:9')
    ->imageResizeTargetWidth('1920')
    ->imageResizeTargetHeight('1080')
```

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')
```

Attachment



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',
    ];

    // ...
}
```

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)
```

You can also enable the re-ordering of uploaded files using the `enableReordering()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
    ->multiple()
    ->enableReordering()
```

You can add a button to open each file in a new tab with the `enableOpen()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
    ->multiple()
    ->enableOpen()
```

If you wish to add a download button to each file instead, you can use the `enableDownload()` method:

```
use Filament\Forms\Components\FileUpload;

FileUpload::make('attachments')
    ->multiple()
    ->enableDownload()
```

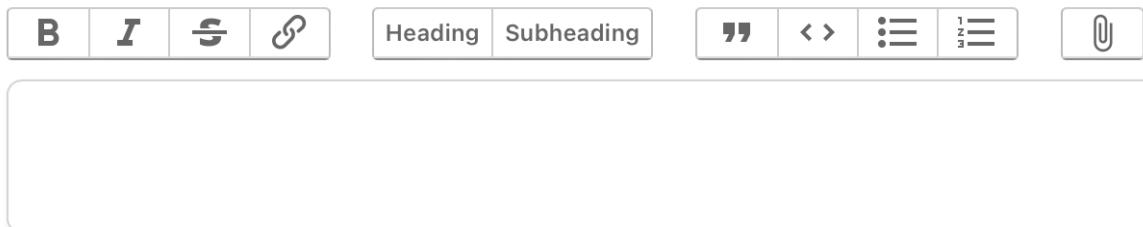
Filament also supports [spatie/laravel-medialibrary](#). See our [plugin documentation](#) for more information.

Rich editor

The rich editor allows you to edit and preview HTML content, as well as upload images.

```
use Filament\Forms\Components\RichEditor;  
  
RichEditor::make('content')
```

Content



You may enable / disable toolbar buttons using a range of convenient methods:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->toolbarButtons([
        'attachFiles',
        'blockquote',
        'bold',
        'bulletList',
        'codeBlock',
        'h2',
        'h3',
        'italic',
        'link',
        'orderedList',
        'redo',
        'strike',
        'underline',
        'undo',
    ])
RichEditor::make('content')
    ->disableToolbarButtons([
        'attachFiles',
        'codeBlock',
    ])
RichEditor::make('content')
    ->disableAllToolbarButtons()
    ->enableToolbarButtons([
        'bold',
        'bulletList',
        'italic',
        'strike',
    ])
])
```

You may customize how images are uploaded using configuration methods:

```
use Filament\Forms\Components\RichEditor;

RichEditor::make('content')
    ->fileAttachmentsDisk('s3')
    ->fileAttachmentsDirectory('attachments')
    ->fileAttachmentsVisibility('private')
```

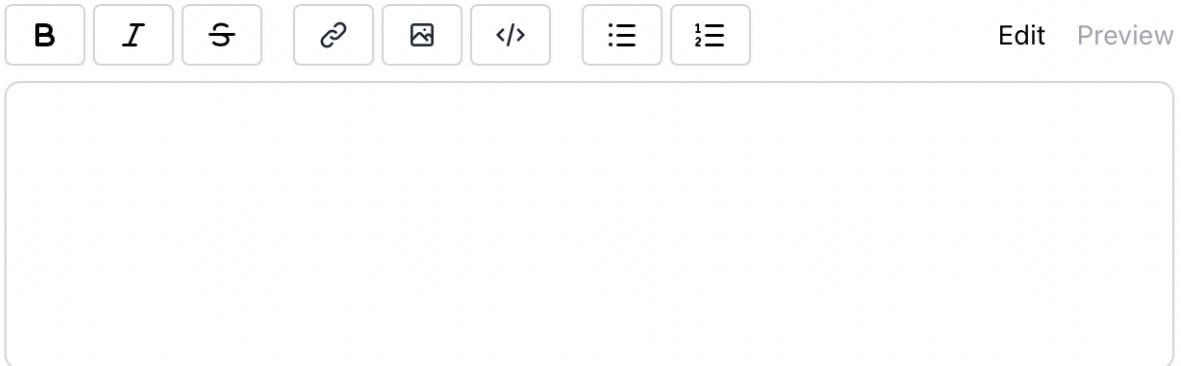
Markdown editor

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')
```

Content



You may enable / disable toolbar buttons using a range of convenient methods:

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
    ->toolbarButtons([
        'attachFiles',
        'bold',
        'bulletList',
        'codeBlock',
        'edit',
        'italic',
        'link',
        'orderedList',
        'preview',
        'strike',
    ])
MarkdownEditor::make('content')
    ->disableToolbarButtons([
        'attachFiles',
        'codeBlock',
    ])
MarkdownEditor::make('content')
    ->disableAllToolbarButtons()
    ->enableToolbarButtons([
        'bold',
        'bulletList',
        'edit',
        'italic',
        'preview',
        'strike',
    ])
)
```

You may customize how images are uploaded using configuration methods:

```
use Filament\Forms\Components\MarkdownEditor;

MarkdownEditor::make('content')
    ->fileAttachmentsDisk('s3')
    ->fileAttachmentsDirectory('attachments')
    ->fileAttachmentsVisibility('private')
```

Hidden

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')
```

Repeater

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 *
Taylor Otwell	Owner
Mohamed Said	Administrator
Dries Vints	Member
+ 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](#).

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)
```

You may set a label to customize the text that should be displayed in the button for adding a repeater item:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
->schema([
    // ...
])
->createItemButtonLabel('Add member')
```

Members

The screenshot shows a 'Members' form component. It displays a single item in a repeater. The item has two fields: 'Name *' containing 'Taylor Otwell' and 'Role *' containing 'Owner'. A red trash icon is located in the top right corner of the item's container. Below the item is a button labeled '+ Add member'.

You may also prevent the user from adding items, deleting items, or moving items inside the repeater:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
->schema([
    // ...
])
->disableItemCreation()
->disableItemDeletion()
->disableItemMovement()
```

You may customize the number of items that may be created, using the `minItems()` and `maxItems()` methods:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
->schema([
    // ...
])
->minItems(1)
->maxItems(10)
```

Collapsible

The repeater may be `collapsible()` to optionally hide content in long forms:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->schema([
        // ...
    ])
    ->collapsible()
```

You may collapse all items by default:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->schema([
        // ...
    ])
    ->collapsed()
```

Cloning items

You may allow repeater items to be duplicated using the `cloneable()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->schema([
        // ...
    ])
    ->cloneable()
```

Populating automatically from a relationship

You may employ the `relationship()` method of the repeater to configure a relationship to automatically retrieve and save repeater items:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
```

To set this functionality up, you must also follow the instructions set out in the [field relationships](#) section. If you're using the [admin panel](#), you can skip this step.

Ordering items

By default, ordering relationship repeater items is disabled. This is because your related model needs an `sort` column to store the order of related records. To enable ordering, you may use the `orderable()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->orderable()
```

This assumes that your related model has a `sort` column.

If you use something like [spatie/eloquent-sortable](#) with an order column such as `order_column`, you may pass this in to `orderable()`:

```
use Filament\Forms\Components\Repeater;

Repeater::make('qualifications')
    ->relationship()
    ->schema([
        // ...
    ])
    ->orderable('order_column')
```

Grid layout

You may organize repeater items into columns by using the `grid()` method:

```
use Filament\Forms\Components\Repeater;

Repeater::make('members')
    ->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.

Item labels

You may add a label for repeater items using the `itemLabel()` method:

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;

Repeater::make('members')
    ->schema([
        TextInput::make('name')
            ->lazy(),
    ])
    ->itemLabel(fn (array $state): ?string => $state['name'] ?? null),
```

Any fields that you use from `$state` should be `reactive()` or `lazy()` if you wish to see the item label update live as you use the form.

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 behaviour 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')`.

Builder

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. 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\MarkdownEditor;
use Filament\Forms\Components>Select;
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(),
]),
Builder\Block::make('paragraph')
->schema([
    MarkdownEditor::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

The screenshot shows the Filament Content builder interface. At the top right are a downward arrow and a trash can icon. Below is a section labeled "Heading *". A large input field is present. Below it is a section labeled "Level *". A dropdown menu with the placeholder "Select an option" is shown. Another downward arrow icon is at the top right of this section. Below these sections is a "Paragraph *" section. It includes a toolbar with icons for bold (B), italic (I), strikethrough (S), link (link icon), image (image icon), code (code icon), align left (align left icon), align center (align center icon), and align right (align right icon). To the right of the toolbar are "Edit" and "Preview" buttons. A large input field for text is below the toolbar. At the bottom is a button labeled "+ 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(),
            // ...
        ]),
        // ...
    ])
])
```

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](#):

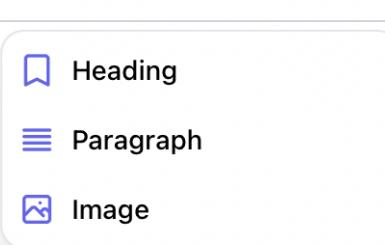
```
use Filament\Forms\Components\Builder;

Builder\Block::make('heading')->label(__('blocks.heading'))
```

Blocks may also have an icon, which is displayed next to the label. The `icon()` method accepts the name of any Blade icon component:

```
use Filament\Forms\Components\Builder;

Builder\Block::make('heading')->icon('heroicon-o-bookmark')
```



You may customize the number of items that may be created, using the `minItems()` and `maxItems()` methods:

```
use Filament\Forms\Components\Builder;
use Filament\Forms\Components\TextInput;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->minItems(1)
    ->maxItems(10)
```

Collapsible

The builder may be `collapsible()` to optionally hide content in long forms:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->collapsible()
```

You may collapse all items by default:

```
use Filament\Forms\Components\Builder;

Builder::make('content')
    ->blocks([
        // ...
    ])
    ->collapsed()
```

Tags input

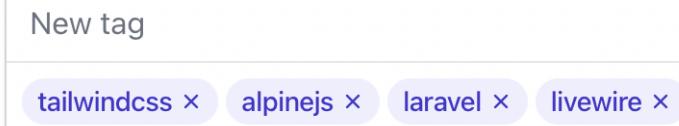
The tags input component allows you to interact with a list of tags.

By default, tags are stored in JSON:

```
use Filament\Forms\Components\TagsInput;

TagsInput::make('tags')
```

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',
    ];

    // ...
}
```

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(',')
```

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',
])
```

Tags

New tag

tailwindcss
alpinejs
laravel
livewire

Filament also supports [spatie/laravel-tags](#). See our [plugin documentation](#) for more information.

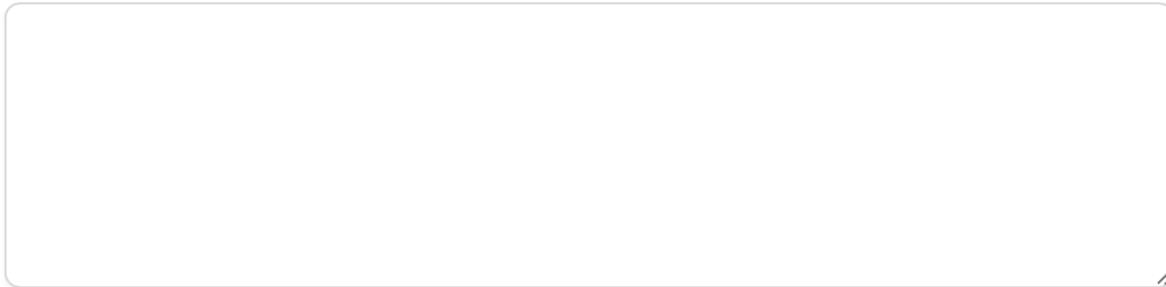
Textarea

The textarea allows you to interact with a multi-line string:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
```

Description



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)
```

You may limit the length of the string by setting the `minLength()` and `maxLength()` methods. These methods add both frontend and backend validation:

```
use Filament\Forms\Components\Textarea;

Textarea::make('description')
->minLength(50)
->maxLength(500)
```

Key-value

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 ·	
og:type	website	
og:site_name	Filament	
+ Add Row		

You may customize the labels for the key and value fields using the `keyLabel()` and `valueLabel()` methods:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
->keyLabel('Property name')
->valueLabel('Property value')
```

Meta

Property name	Property value

+ Add Row

You may also prevent the user from adding rows, deleting rows, or editing keys:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
    ->disableAddingRows()
    ->disableDeletingRows()
    ->disableEditingKeys()
```

You can allow the user to reorder rows within the table:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
    ->reorderable()
```

You may also add placeholders for the key and value fields using the `keyPlaceholder()` and `valuePlaceholder()` methods:

```
use Filament\Forms\Components\KeyValue;

KeyValue::make('meta')
    ->keyPlaceholder('Property name')
    ->valuePlaceholder('Property value')
```

Meta

Key	Value
Property name	Property value
+ Add Row	

Color picker

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')
```

Color

#4f46e5



Alternatively, you can use a different format:

```
use Filament\Forms\Components\ColorPicker;

ColorPicker::make('hs1_color')->hs1()
ColorPicker::make('rgb_color')->rgb()
ColorPicker::make('rgba_color')->rgba()
```

View

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('notifications')->view('filament.forms.components.range-slider')
```

Inside your view, you may interact with the state of the form component using Livewire and Alpine.js.

The `$getStatePath()` closure may be used by the view to retrieve the Livewire property path of the field. You could use this to `wire:model` a value, or `$wire->entangle` it with Alpine.js.

Using [Livewire's entangle](#) allows sharing state with Alpine.js:

```
<x-dynamic-component
    :component="$getFieldWrapperView()"
    :id="$getId()"
    :label="$getLabel()"
    :label-sr-only="$isLabelHidden()"
    :helper-text="$getHelperText()"
    :hint="$getHint()"
    :hint-action="$getHintAction()"
    :hint-color="$getHintColor()"
    :hint-icon="$getHintIcon()"
    :required="$isRequired()"
    :state-path="$getStatePath()"
>
    <div x-data="{ state: $wire.entangle('{{ $getStatePath() }}').defer }">
        <!-- Interact with the `state` property in Alpine.js -->
    </div>
</x-dynamic-component>
```

Or, you may bind the value to a Livewire property using `wire:model`:

```
<x-dynamic-component
    :component="$getFieldWrapperView()"
    :id="$getId()"
    :label="$getLabel()"
    :label-sr-only="$isLabelHidden()"
    :helper-text="$getHelperText()"
    :hint="$getHint()"
    :hint-action="$getHintAction()"
    :hint-color="$getHintColor()"
    :hint-icon="$getHintIcon()"
    :required="$isRequired()"
    :state-path="$getStatePath()"
>
    <input wire:model.defer="{{ $getStatePath() }}" />
</x-dynamic-component>
```

Building custom fields

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';
}
```

Inside your view, you may interact with the state of the form component using Livewire and Alpine.js.

The `$getStatePath()` closure may be used by the view to retrieve the Livewire property path of the field. You could use this to `wire:model` a value, or `$wire.entangle` it with Alpine.js:

```
<x-dynamic-component
    :component="$getFieldWrapperView()"
    :id="$getId()"
    :label="$getLabel()"
    :label-sr-only="$isLabelHidden()"
    :helper-text="$getHelperText()"
    :hint="$getHint()"
    :hint-action="$getHintAction()"
    :hint-color="$getHintColor()"
    :hint-icon="$getHintIcon()"
    :required="$isRequired()"
    :state-path="$getStatePath()"
>
    <div x-data="{ state: $wire.entangle('{{ $getStatePath() }}').defer }">
        <!-- Interact with the `state` property in Alpine.js -->
    </div>
</x-dynamic-component>
```

Layout

Getting started

Layout component classes can be found in the `Filament\Form\Components` namespace.

They reside within the schema of your form, alongside any [fields](#).

If you're using the layout components in a Livewire component, you can put them in the `getFormSchema()` method:

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

If you're using them in admin panel resources or relation managers, you must put them in the `$form->schema()` method:

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

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()
    ->schema([
        // ...
    ])
```

Columns

You may create multiple columns within each layout component using the `columns()` method:

```
use Filament\Forms\Components\Card;

Card::make() ->columns(2)
```

For more information about creating advanced, responsive column layouts, please see the [grid section](#). All column options in that section are also available in other layout components.

Controlling field column span

You may specify the number of columns that any component may span in the parent grid:

```
use Filament\Forms\Components\Grid;
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\TextInput;

Grid::make(3)
->schema([
    TextInput::make('name')
        ->columnSpan(2),
    // ...
])
```

You may use `columnSpan('full')` to ensure that a column spans the full width of the parent grid, however many columns it has:

```
use Filament\Forms\Components\Grid;
use Filament\Forms\Components\RichEditor;
use Filament\Forms\Components\TextInput;

Grid::make(3)
->schema([
    TextInput::make('name')
        ->columnSpan('full'),
    // ...
])
```

Instead, you can even define how many columns a component may consume at any breakpoint:

```
use Filament\Forms\Components\Grid;
use Filament\Forms\Components\TextInput;

Grid::make([
    'default' => 1,
    'sm' => 3,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    TextInput::make('name')
        ->columnSpan([
            'sm' => 2,
            'xl' => 3,
            '2xl' => 4,
        ]),
    // ...
])
```

Setting an ID

You may define an ID for the component using the `id()` method:

```
use Filament\Forms\Components\Card;

Card::make()->id('main-card')
```

Custom attributes

The HTML of components can be customized even further, by passing an array of `extraAttributes()`:

```
use Filament\Forms\Components\Card;

Card::make() ->extraAttributes(['class' => 'bg-gray-50'])
```

Global settings

If you wish to change the default behaviour 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 card components have 2 columns by default, you can do it like so:

```
use Filament\Forms\Components\Card;

Card::configureUsing(function (Card $card): void {
    $card->columns(2);
});
```

Of course, you are still able to overwrite this on each field individually:

```
use Filament\Forms\Components\Card;

Card::make() ->columns(1)
```

Saving data to relationships

You may load and save the contents of a layout component to a `HasOne`, `BelongsTo` or `MorphOne` Eloquent relationship, using the `relationship()` method:

```
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.

To set this functionality up, you must also follow the instructions set out in the [field relationships](#) section. If you're using the [admin panel](#), you can skip this step.

Grid

Generally, form fields are stacked on top of each other in one column. To change this, you may use a grid component:

```
use Filament\Forms\Components\Grid;

Grid::make()
->schema([
    // ...
])
```

By default, grid components will create a two column grid for the Tailwind `md` breakpoint and higher.

You may pass a different number of columns to the grid's `md` breakpoint:

```
use Filament\Forms\Components\Grid;

Grid::make(3)
->schema([
    // ...
])
```

To customize the number of columns in any grid at different breakpoints, you may pass an array of breakpoints and columns:

```
use Filament\Forms\Components\Grid;

Grid::make([
    'default' => 1,
    'sm' => 2,
    'md' => 3,
    'lg' => 4,
    'xl' => 6,
    '2xl' => 8,
])
->schema([
    // ...
])
```

Since Tailwind is mobile-first, if you leave out a breakpoint, it will fall back to the one set below it:

```
use Filament\Forms\Components\Grid;

Grid::make([
    'sm' => 2,
    'xl' => 6,
])
->schema([
    // ...
])
```

Fieldset

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([
    // ...
])
```

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

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('Heading')
->tabs([
    Tabs\Tab::make('Label 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Label 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Label 3')
        ->schema([
            // ...
        ]),
])
```

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('Heading')
->tabs([
    Tabs\Tab::make('Label 1')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Label 2')
        ->schema([
            // ...
        ]),
    Tabs\Tab::make('Label 3')
        ->schema([
            // ...
        ]),
])
->activeTab(2)
```

Tabs may have an icon and badge, which you can set using the `icon()` and `badge()` methods:

```
use Filament\Forms\Components\tabs;

Tabs::make('Heading')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-o-bell') // [tl! focus:start]
        ->badge('39') // [tl! focus:end]
        ->schema([
            // ...
        ]),
    // ...
])
```

Icons can be modified using the `iconPosition()` and `iconColor()` methods:

```
use Filament\Forms\Components\tabs;

Tabs::make('Heading')
->tabs([
    Tabs\Tab::make('Notifications')
        ->icon('heroicon-o-bell')
        ->iconPosition('after') // `before` or `after` [tl! focus:end]
        ->iconColor('success') // `danger`, `primary`, `success`, `warning` or `secondary`
[tl! focus:end]
        ->schema([
            // ...
        ]),
    // ...
])
```

Wizard

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([
            // ...
        ]),
])
```

We have different setup instructions if you're looking to add a wizard to an admin panel [resource Create page](#) or a table [action](#). Following that documentation will ensure that the ability to submit the form is only available on the last step.

Each step has a mandatory label. You may optionally also add a description for extra detail:

```
use Filament\Forms\Components\Wizard;

Wizard\Step::make('Order')
    ->description('Review your basket')
    ->schema([
        // ...
]),
```

Steps may also have an icon, which can be the name of any Blade icon component:

```
use Filament\Forms\Components\Wizard;

Wizard\Step::make('Order')
    ->icon('heroicon-o-shopping-bag')
    ->schema([
        // ...
]),
```

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>'))
```

You may use the `startOnStep()` method to load a specific step in the wizard:

```
use Filament\Forms\Components\Wizard;

Wizard::make([
    // ...
])->startOnStep(2)
```

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()
```

Section

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('Heading')
    ->description('Description')
    ->schema([
        // ...
    ])
```

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)
```

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('Heading')
    ->description('Description')
    ->aside()
    ->schema([
        // ...
    ])
```

Sections may be `collapsible()` to optionally hide content in long forms:

```
use Filament\Forms\Components\Section;

Section::make('Heading')
    ->schema([
        // ...
    ])
    ->collapsible()
```

Your sections may be `collapsed()` by default:

```
use Filament\Forms\Components\Section;

Section::make('Heading')
    ->schema([
        // ...
    ])
    ->collapsed()
```

When nesting sections, you can use a more compact styling:

```
use Filament\Forms\Components\Section;

Section::make('Heading')
    ->schema([
        // ...
    ])
    ->compact()
```

Placeholder

Placeholders can be used to render text-only "fields" within your forms. Each placeholder has `content()`, which cannot be changed by the user.

Important: All fields require a unique name. That also applies to Placeholders!

```
use Filament\Forms\Components\Placeholder;

Placeholder::make('Label')
    ->content('Content, displayed underneath the label')
```

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>'))
```

Card

The card component may be used to render the form components inside a card:

```
use Filament\Forms\Components\Card;

Card::make()
->schema([
    // ...
])
```

You may use the `columns()` method to easily create a grid within the card:

```
use Filament\Forms\Components\Card;

Card::make()
->schema([
    // ...
])
->columns(2)
```

Inline labels

You may use the `inlineLabel()` method to make the form labels and fields in separate columns, inline with each other. It works on all layout components, each field inside will have an inline label.

```
use Filament\Forms\Components\Card;

Card::make()
->schema([
    // ...
])
->inlineLabel()
```

View

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')
```

Inside your view, you may render the component's `schema()` using the `$getComponentContainer()` closure:

```
<div>
{{ $getChildComponentContainer() }}
</div>
```

Building custom layout components

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 new static();
    }
}
```

Inside your view, you may render the component's `schema()` using the `$getChildComponentContainer()` closure:

```
<div>
{{ $getChildComponentContainer() }}
</div>
```

Validation

Getting started

Validation rules may be added to any [field](#).

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('startDate')->after('tomorrow')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('startDate')
Field::make('endDate')->after('startDate')
```

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('startDate')->afterOrEqual('tomorrow')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('startDate')
Field::make('endDate')->afterOrEqual('startDate')
```

Alpha

The field must be entirely alphabetic characters. [See the Laravel documentation](#).

```
Field::make('name')->alpha()
```

Alpha Dash

The field may have alpha-numeric characters, as well as dashes and underscores. [See the Laravel documentation.](#)

```
Field::make('name')->alphaDash()
```

Alpha Numeric

The field must be entirely alpha-numeric 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('startDate')->before('first day of next month')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('startDate')->before('endDate')
Field::make('endDate')
```

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('startDate')->beforeOrEqual('end of this month')
```

Alternatively, you may pass the name of another field to compare against:

```
Field::make('startDate')->beforeOrEqual('endDate')
Field::make('endDate')
```

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('backupEmail')->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(callback: 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')
```

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()
```

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 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 of 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 of 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(ignorable: $ignoredUser)
```

If you're using the [admin panel](#), 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 `callback` parameter:

```
use Illuminate\Validation\Rules\Unique;

Field::make('email')
    ->unique(callback: function (Unique $rule) {
        return $rule->where('is_active', 1);
    })
}
```

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](#):

```
TextInput::make('slug')->rules([
    function () {
        return function (string $attribute, $value, Closure $fail) {
            if ($value === 'foo') {
                $fail('The :attribute is invalid.');
            }
        };
    },
])
```

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')
```

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 admin panel and you want this behaviour 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();
};
```

Advanced

Using closure customization

All configuration methods for [fields](#) and [layout components](#) accept closures 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 () {
        if ($auth()->user()->country_id === 'us') {
            return 'm/d/Y';
        } else {
            return 'd/m/Y';
        }
    })
}

Select::make('userId')
    ->options(function () {
        return User::all()->pluck('name', 'id');
    })
}

TextInput::make('middle_name')
    ->required(function () {
        return $auth()->user()->hasMiddleName();
    })
}
```

This alone unlocks many customization possibilities.

The package is also able to inject many utilities to use inside these closures, as parameters.

If you wish to access the current state (value) of the component, define a `$state` parameter:

```
function ($state) {
    // ...
}
```

If you wish to access the current component instance, define a `$component` parameter:

```
use Filament\Forms\Components\Component;

function (Component $component) {
    // ...
}
```

If you wish to access the current Livewire component instance, define a `$livewire` parameter:

```
use Livewire\Component as Livewire;

function (Livewire $livewire) {
    // ...
}
```

If you have defined a form or component Eloquent model instance, define a `$record` parameter:

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

function (?Model $record) {
    // ...
}
```

You may also retrieve the value of another field from within a callback, using a closure `$get` parameter:

```
use Closure;

function (Closure $get) {
    $email = $get('email'); // Store the value of the `email` field in the `$email` variable.
    //...
}
```

In a similar way to `$get`, you may also set the value of another field from within a callback, using a closure `$set` parameter:

```
use Closure;

function (Closure $set) {
    $set('title', 'Blog Post'); // Set the `title` field to `Blog Post`.
    //...
}
```

If you're writing a form for an admin panel resource or relation manager, and you wish to check if a form is `create`, `edit` or `view`, use the `$context` parameter:

```
function (string $context) {
    // ...
}
```

Outside of the admin panel, you can set a form's context by defining a `getFormContext()` method on your Livewire component.

Callbacks are evaluated using Laravel's `app() ->call()` under the hood, so you are able to combine multiple parameters in any order:

```
use Closure;
use Livewire\Component as Livewire;

function (Livewire $livewire, Closure $get, Closure $set) {
    // ...
}
```

Reloading the form when a field is updated

By default, forms are only reloaded when they are validated or submitted. You may allow a form to be reloaded when a field is changed, by using the `reactive()` method on that field:

```
use Filament\Forms\Components\TextInput;

TextInput::make('title')->reactive()
```

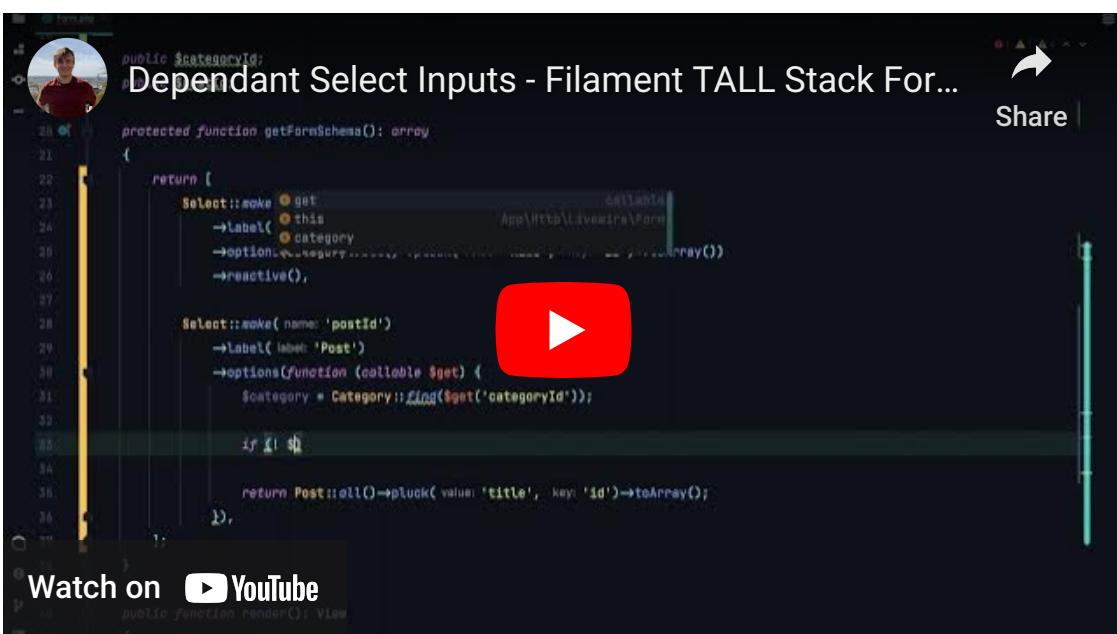
A great example to give a use case for this is when you wish to generate a slug from a title field automatically:



Dependant fields / components

You may use the techniques described in the [closure customization section](#) to build completely dependant fields and components, with full control over customization based on the values of other fields in your form.

For example, you can build dependant `select` inputs:



Sometimes, you may wish to conditionally hide any form component based on the value of a field. You may do this with a `hidden()` method:

```
use Closure;
use Filament\Forms\Components\TextInput;

TextInput::make('newPassword')
    ->password()
    ->reactive()

TextInput::make('newPasswordConfirmation')
    ->password()
    ->hidden(fn (Closure $get) => $get('newPassword') === null)
```

The field/s you're depending on should be `reactive()`, to ensure the Livewire component is reloaded when they are updated.

Field lifecycle

Hydration

Hydration is the process which fill 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()`.

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')
    ->afterStateHydrated(function (TextInput $component, $state) {
        $component->state(ucwords($state));
    })
```

Updates

You may use the `afterStateUpdated()` method to customize what happens after a field is updated.

In this example, the `slug` field is updated with the slug version of the `title` field automatically:

```
use Closure;
use Filament\Forms\Components\TextInput;
use Illuminate\Support\Str;

TextInput::make('title')
    ->reactive()
    ->afterStateUpdated(function (Closure $set, $state) {
        $set('slug', Str::slug($state));
    })
TextInput::make('slug')
```

Dehydration

Dehydration is the process which gets data from fields, and transforms it. It runs when you call the `form's getState()` method. You may customize how the state is dehydrated from the form by returning the transformed state from the

```
dehydrateStateUsing() callback.
```

In this example, the `name` field will always be dehydrated with the correctly capitalized name:

```
use Filament\Forms\Components\TextInput;

TextInput::make('name')->dehydrateStateUsing(fn ($state) => ucwords($state))
```

You may also prevent the field from being dehydrated altogether by passing `false` to `dehydrated()`.

In this example, the `passwordConfirmation` field will not be present in the array returned from `getData()`:

```
use Filament\Forms\Components\TextInput;

TextInput::make('passwordConfirmation')
->password()
->dehydrated(false)
```

Using form events

Forms can dispatch and listen to events, which allow the frontend and backend to communicate.

These events can be dispatched in a component view, and then listened to by a component's class.

Dispatching events

To dispatch a form event, call the `dispatchFormEvent()` Livewire method with the event name:

```
<button wire:click="dispatchFormEvent('save')">
    Save
</button>
```

Usually, you will want to dispatch events for a specific component class. In this case, you should prefix the event name with the component name, and pass the component's state path as a second parameter:

```
<button wire:click="dispatchFormEvent('repeater::createItem', '{{ $getStatePath() }}')">
    Add item
</button>
```

You may also pass other parameters to the event:

```
<button wire:click="dispatchFormEvent('repeater::deleteItem', '{{ $getStatePath() }}', '{{ $uuid }}')">
    Delete item
</button>
```

Listening to events

You may register listeners for form events by calling the `registerListeners()` method when your component is `setUp()`:

```
use Filament\Forms\Components\Component;

protected function setUp(): void
{
    parent::setUp();

    $this->registerListeners([
        'save' => [
            function (Component $component): void {
                // ...
            },
        ],
    ]);
}
```

If your event is component-specific, you'll want to ensure that the component is not disabled, and that the component's state path matches the event's second parameter:

```
use Filament\Forms\Components\Component;

protected function setUp(): void
{
    parent::setUp();

    $this->registerListeners([
        'repeater::createItem' => [
            function (Component $component, string $statePath): void {
                if ($component->isDisabled()) {
                    return;
                }

                if ($statePath !== $component->getStatePath()) {
                    return;
                }

                // ...
            },
        ],
    ]);
}
```

Additionally, you may receive other parameters from the event:

```
use Filament\Forms\Components\Component;

protected function setUp(): void
{
    parent::setUp();

    $this->registerListeners([
        'repeater::deleteItem' => [
            function (Component $component, string $statePath, string $uuidToDelete): void {
                if ($component->isDisabled()) {
                    return;
                }

                if ($statePath !== $component->getStatePath()) {
                    return;
                }

                // Delete item with UUID `$uuidToDelete`
            },
        ],
    ]);
}
```

Testing

All examples in this guide will be written using [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(),
    // ...
]);
```

Note that if you have multiple forms on a Livewire component, you can specify which form you want to fill using

`fillForm([...], 'createForm')`.

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),
        ]);
});
```

Note that if you have multiple forms on a Livewire component, you can specify which form you want to check using

`assertFormSet([...], 'createForm')`.

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,
        ])
        ->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('save')
    ->assertHasNoFormErrors();
```

Note that 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();
});
```

Note that 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();
        });
});
```

Note that 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', 'createForm')`.

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');
});
```

Note that 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', 'createForm')`.

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');
});
```

Note that 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')`.

Chapter 3

Table Builder

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

The table builder comes pre-installed inside the [admin panel 2.x](#), but you must still follow the installation instructions below if you're using it in the rest of your app.

First, require the table builder using Composer:

```
composer require filament/tables:"^2.0"
```

New Laravel projects

To get started with the table builder quickly, you can set up [Livewire](#), [Alpine.js](#) and [Tailwind CSS](#) with these commands:

```
php artisan tables:install
npm install
npm run dev
```

These commands will ruthlessly overwrite existing files in your application, hence why we only recommend using this method for new projects.

You're now ready to start [building tables!](#)

Existing Laravel projects

The package uses the following dependencies:

- [Alpine.js](#)
- [Alpine.js Focus Plugin](#)
- [PostCSS](#)
- [Tailwind CSS](#)
- [Tailwind CSS Forms plugin](#)
- [Tailwind CSS Typography plugin](#)

You may install these through NPM:

```
npm install alpinejs @alpinejs/focus postcss tailwindcss @tailwindcss/forms
@tailwindcss/typography --save-dev
```

Configuring Tailwind CSS

To finish installing Tailwind, you must create a new `tailwind.config.js` file in the root of your project. The easiest way to do this is by running `npx tailwindcss init`.

In `tailwind.config.js`, register the plugins you installed, and add custom colors used by the table builder:

```

import colors from 'tailwindcss/colors' // [tl! focus:start]
import forms from '@tailwindcss/forms'
import typography from '@tailwindcss/typography' // [tl! focus:end]

export default {
    content: [
        './resources/**/*.blade.php',
        './vendor/filament/**/*.blade.php', // [tl! focus]
    ],
    theme: {
        extend: {
            colors: { // [tl! focus:start]
                danger: colors.rose,
                primary: colors.blue,
                success: colors.green,
                warning: colors.yellow,
            }, // [tl! focus:end]
        },
    },
    plugins: [
        forms, // [tl! focus:start]
        typography, // [tl! focus:end]
    ],
}

```

Of course, you may specify your own custom `primary`, `success`, `warning` and `danger` colors, which will be used instead.

Bundling assets

New Laravel projects use Vite for bundling assets by default. However, your project may still use Laravel Mix. Read the steps below for the bundler used in your project.

Vite

If you're using Vite, you should manually install [Autoprefixer](#) through NPM:

```
npm install autoprefixer --save-dev
```

Create a `postcss.config.js` file in the root of your project, and register Tailwind CSS and Autoprefixer as plugins:

```

export default {
    plugins: {
        tailwindcss: {},
        autoprefixer: {},
    },
}

```

You may also want to update your `vite.config.js` file to refresh the page after Livewire components or custom table columns have been updated:

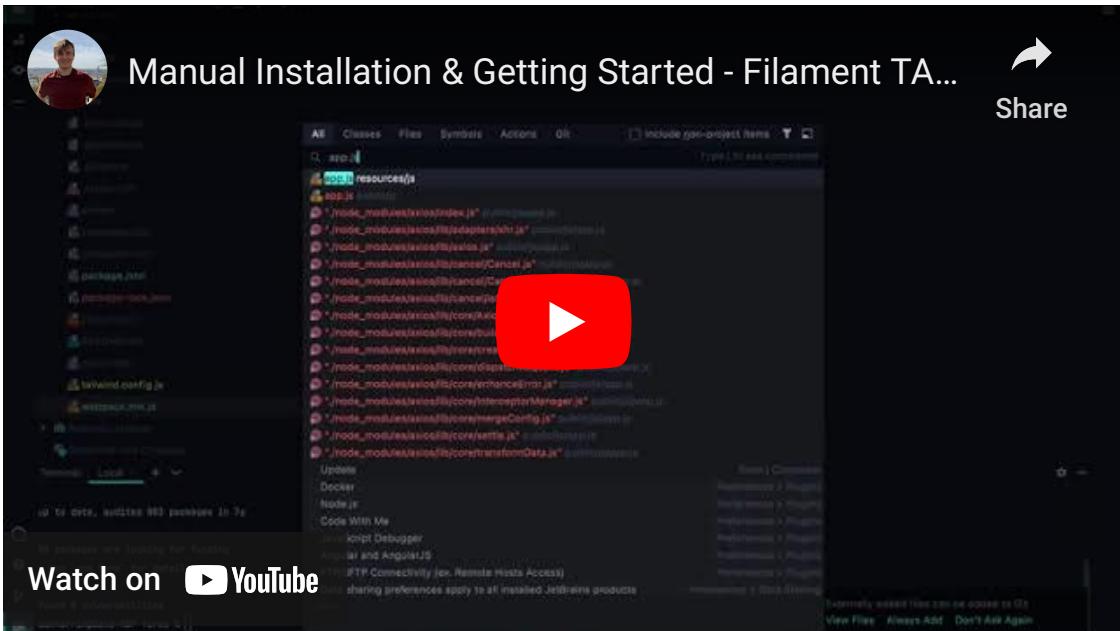
```

import { defineConfig } from 'vite';
import laravel, { refreshPaths } from 'laravel-vite-plugin'; // [tl! focus]

export default defineConfig({
  plugins: [
    laravel({
      input: [
        'resources/css/app.css',
        'resources/js/app.js',
      ],
      refresh: [ // [tl! focus:start]
        ...refreshPaths,
        'app/Http/Livewire/**',
        'app/Tables/Columns/**',
      ], // [tl! focus:end]
    }),
  ],
});

```

Laravel Mix



In your `webpack.mix.js` file, register Tailwind CSS as a PostCSS plugin:

```

const mix = require('laravel-mix')

mix.js('resources/js/app.js', 'public/js')
  .postCss('resources/css/app.css', 'public/css', [
    require('tailwindcss'), // [tl! focus]
  ])

```

Configuring styles

In `/resources/css/app.css`, import `filament/forms` vendor CSS and Tailwind CSS:

```
@import '../../../../../vendor/filament/forms/dist/module.esm.css';

@tailwind base;
@tailwind components;
@tailwind utilities;
```

Configuring scripts

In `/resources/js/app.js`, import `Alpine.js`, `@alpinejs/focus`, the `filament/forms` and `filament/notifications` plugins, and register them:

```
import Alpine from 'alpinejs'
import Focus from '@alpinejs/focus'
import FormsAlpinePlugin from '../../../../../vendor/filament/forms/dist/module.esm'
import NotificationsAlpinePlugin from '../../../../../vendor/filament/notifications/dist/module.esm'

Alpine.plugin(Focus)
Alpine.plugin(FormsAlpinePlugin)
Alpine.plugin(NotificationsAlpinePlugin)

window.Alpine = Alpine

Alpine.start()
```

Compiling assets

Compile your new CSS and JS assets using `npm run dev`.

Configuring layout

Finally, create a new `resources/views/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>
        @vite(['resources/css/app.css', 'resources/js/app.js'])
        @livewireStyles
        @livewireScripts
        @stack('scripts')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @livewire('notifications')
    </body>
</html>

```

You're now ready to start [building tables!](#)

Publishing configuration

If you wish, you may publish the configuration of the package using:

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

Publishing translations

If you wish to translate the package, you may publish the language files using:

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

Since this package depends on other Filament packages, you may wish to translate those as well:

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

Upgrading

To upgrade the package to the latest version, you must run:

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

We recommend adding the `filament:upgrade` command to your `composer.json`'s `post-update-cmd` to run it automatically:

```
"post-update-cmd": [
    // ...
    "@php artisan filament:upgrade"
],
```

Getting Started

Preparing your Livewire component

Implement the `HasTable` interface and use the `InteractsWithTable` trait:

```
<?php

namespace App\Http\Livewire;

use Filament\Tables;
use Illuminate\Contracts\View\View;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable // [tl! focus]
{
    use Tables\Concerns\InteractsWithTable; // [tl! focus]

    public function render(): View
    {
        return view('list-posts');
    }
}
```

In your Livewire component's view, render the table:

```
<div>
{{ $this->table }}
</div>
```

Next, add the Eloquent query you would like the table to be based upon in the `getTableQuery()` method:

```
<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    protected function getTableQuery(): Builder // [tl! focus:start]
    {
        return Post::query();
    } // [tl! focus:end]

    public function render(): View
    {
        return view('list-posts');
    }
}
```

Finally, add any columns, filters, and actions to the Livewire component:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Collection;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    protected function getTableQuery(): Builder
    {
        return Post::query();
    }

    protected function getTableColumns(): array // [tl! focus:start]
    {
        return [ // [tl! collapse:start]
            Tables\Columns\ImageColumn::make('author.avatar')
                ->size(40)
                ->circular(),
            Tables\Columns\TextColumn::make('title'),
            Tables\Columns\TextColumn::make('author.name'),
            Tables\Columns\BadgeColumn::make('status')
                ->colors([
                    'danger' => 'draft',
                    'warning' => 'reviewing',
                    'success' => 'published',
                ]),
            Tables\Columns\IconColumn::make('is_featured')->boolean(),
        ]; // [tl! collapse:end]
    }

    protected function getTableFilters(): array
    {
        return [ // [tl! collapse:start]
            Tables\Filters\Filter::make('published')
                ->query(fn (Builder $query): Builder => $query->where('is_published', true)),
            Tables\Filters\SelectFilter::make('status')
                ->options([
                    'draft' => 'Draft',
                    'in_review' => 'In Review',
                    'approved' => 'Approved',
                ]),
        ]; // [tl! collapse:end]
    }

    protected function getTableActions(): array
    {
        return [ // [tl! collapse:start]
            Tables\Actions\Action::make('edit')

```

```
    ->url(fn (Post $record): string => route('posts.edit', $record)),
]; // [tl! collapse:end]
}

protected function getTableBulkActions(): array
{
    return [ // [tl! collapse:start]
        Tables\Actions\BulkAction::make('delete')
            ->label('Delete selected')
            ->color('danger')
            ->action(function (Collection $records): void {
                $records->each->delete();
            })
            ->requiresConfirmation(),
    ]; // [tl! collapse:end]
} // [tl! focus:end]

public function render(): View
{
    return view('list-posts');
}
}
```

Visit your Livewire component in the browser, and you should see the table.

Pagination

By default, tables will be paginated. To disable this, you should override the `isTablePaginationEnabled()` method on your Livewire component:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    protected function getTableQuery(): Builder
    {
        return Post::query();
    }

    protected function getTableColumns(): array
    {
        return [
            Tables\Columns\TextColumn::make('title'),
            Tables\Columns\TextColumn::make('author.name'),
        ];
    }

    protected function isTablePaginationEnabled(): bool // [tl! focus:start]
    {
        return false;
    } // [tl! focus:end]

    public function render(): View
    {
        return view('list-posts');
    }
}

```

You may customize the options for the paginated records per page select by overriding the `getTableRecordsPerPageSelectOptions()` method on your Livewire component:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    protected function getTableQuery(): Builder
    {
        return Post::query();
    }

    protected function getTableColumns(): array
    {
        return [
            Tables\Columns\TextColumn::make('title'),
            Tables\Columns\TextColumn::make('author.name'),
        ];
    }

    protected function getTableRecordsPerPageSelectOptions(): array // [tl! focus:start]
    {
        return [10, 25, 50, 100];
    } // [tl! focus:end]

    public function render(): View
    {
        return view('list-posts');
    }
}

```

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 `getTableQueryStringIdentifier()` on your component, to return a unique query string identifier for that table:

```

protected function getTableQueryStringIdentifier(): string
{
    return 'users';
}

```

Simple pagination

You may use simple pagination by overriding `paginateTableQuery()` method on your Livewire component:

```
use Illuminate\Contracts\Pagination\Paginator;
use Illuminate\Database\Eloquent\Builder;

protected function paginateTableQuery(Builder $query): Paginator
{
    return $query->simplePaginate($this->getTableRecordsPerPage() == -1 ? $query->count() :
    $this->getTableRecordsPerPage());
}
```

Searching records with Laravel Scout

While Filament doesn't provide a direct integration with [Laravel Scout](#), you may override methods to integrate it with your Livewire component.

First, you must ensure that the table search input is visible:

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

Now, use a `whereIn()` clause to filter the query for Scout results:

```
use App\Models\Post;
use Illuminate\Database\Eloquent\Builder;

protected function applySearchToTableQuery(Builder $query): Builder
{
    if ($filled($searchQuery = $this->getTableSearchQuery())) {
        $query->whereIn('id', Post::search($searchQuery)->keys());
    }

    return $query;
}
```

Scout uses this `whereIn()` method to retrieve results internally, so there is no performance penalty for using it.

Clickable rows

Record URLs

You may allow table rows to be completely clickable by overriding the `getTableRecordUrlUsing()` method on your Livewire component:

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

protected function getTableRecordUrlUsing(): ?Closure
{
    return fn (Model $record): string => route('posts.edit', ['record' => $record]);
}
```

In this example, clicking on each post will take you to the `posts.edit` route.

If you'd like to [override the URL](#) for a specific column, or instead [run a Livewire action](#) when a column is clicked, see the [columns documentation](#).

Record actions

Alternatively, you may configure table rows to trigger an action instead of opening a URL:

```
use Closure;
use Filament\Tables\Actions\Action;
use Filament\Tables\Actions\DeleteAction;

protected function getTableRecordActionUsing(): ?Closure
{
    return fn (): string => 'edit';
}
```

In this case, if an `EditAction` or another action with the name `edit` exists on the table row, that will be called. If not, a Livewire public method with the name `edit()` will be called, and the selected record will be passed.

Disabling clickable rows

If you'd like to completely disable the click action for the entire row, you may override the `getTableRecordActionUsing()` method on your Livewire component, and return `null`:

```
use Closure;

protected function getTableRecordActionUsing(): ?Closure
{
    return null;
}
```

Record 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 `getTableRecordClassesUsing()` method:

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

protected function getTableRecordClassesUsing(): ?Closure
{
    return fn (Model $record) => match ($record->status) {
        'draft' => 'opacity-30',
        'reviewing' => [
            'border-l-2 border-orange-600',
            'dark:border-orange-300' => config('tables.dark_mode'),
        ],
        'published' => 'border-l-2 border-green-600',
        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 in your desired PHP files:

```
export default {
    content: ['./app/Filament/**/*.php'],
}
```

Alternatively, you may add the classes to your safelist:

```
export default {
    safelist: [
        'border-green-600',
        'border-l-2',
        'border-orange-600',
        'dark:border-orange-300',
        'opacity-30',
    ],
}
```

Empty state

By default, an "empty state" card will be rendered when the table is empty. To customize this, you may define methods on your Livewire component:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Collection;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    protected function getTableQuery(): Builder
    {
        return Post::query();
    }

    protected function getTableColumns(): array
    {
        return [ // [tl! collapse:start]
            Tables\Columns\ImageColumn::make('author.avatar')
                ->size(40)
                ->circular(),
            Tables\Columns\TextColumn::make('title'),
            Tables\Columns\TextColumn::make('author.name'),
            Tables\Columns\BadgeColumn::make('status')
                ->colors([
                    'danger' => 'draft',
                    'warning' => 'reviewing',
                    'success' => 'published',
                ]),
            Tables\Columns\IconColumn::make('is_featured')->boolean(),
        ]; // [tl! collapse:end]
    }

    protected function getTableEmptyStateIcon(): ?string // [tl! focus:start]
    {
        return 'heroicon-o-bookmark';
    }

    protected function getTableEmptyStateHeading(): ?string
    {
        return 'No posts yet';
    }

    protected function getTableEmptyStateDescription(): ?string
    {
        return 'You may create a post using the button below.';
    }

    protected function getTableEmptyStateActions(): array
    {
        return [

```

```

Tables\Actions\Action::make('create')
    ->label('Create post')
    ->url(route('posts.create'))
    ->icon('heroicon-o-plus')
    ->button(),
];
} // [tl! focus:end]

public function render(): View
{
    return view('list-posts');
}
}

```

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:

```

protected $queryString = [
    'tableFilters',
    'tableSortColumn',
    'tableSortDirection',
    'tableSearchQuery' => ['except' => ''],
    'tableColumnSearchQueries',
];

```

Reordering records

To allow the user to reorder records using drag and drop in your table, you can use the `getTableReorderColumn()` method:

```

protected function getTableReorderColumn(): ?string
{
    return 'sort';
}

```

When making the table reorderable, a new button will be available on the table to toggle reordering.

The `getTableReorderColumn()` method returns 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 return this instead:

```

protected function getTableReorderColumn(): ?string
{
    return 'order_column';
}

```

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:

```
protected function isTablePaginationEnabledWhileReordering(): bool
{
    return true;
}
```

Polling content

You may poll table content so that it refreshes at a set interval, using the `getTablePollingInterval()` method:

```
protected function getTablePollingInterval(): ?string
{
    return '10s';
}
```

Using the form builder

Internally, the table builder uses the [form builder](#) to implement filtering, actions, and bulk actions. Because of this, the form builder is already set up on your Livewire component and ready to use with your own custom forms.

You may use the default `form` out of the box:

```

<?php

namespace App\Http\Livewire;

use App\Models\Post;
use Filament\Tables;
use Illuminate\Contracts\View\View;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Collection;
use Livewire\Component;

class ListPosts extends Component implements Tables\Contracts\HasTable
{
    use Tables\Concerns\InteractsWithTable;

    public function mount(): void
    {
        $this->form->fill();
    }

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

    protected function getTableQuery(): Builder // [tl! collapse:start]
    {
        return Post::query();
    }

    protected function getTableColumns(): array
    {
        return [
            Tables\Columns\ImageColumn::make('author.avatar')
                ->size(40)
                ->circular(),
            Tables\Columns\TextColumn::make('title'),
            Tables\Columns\TextColumn::make('author.name'),
            Tables\Columns\BadgeColumn::make('status')
                ->colors([
                    'danger' => 'draft',
                    'warning' => 'reviewing',
                    'success' => 'published',
                ]),
            Tables\Columns\IconColumn::make('is_featured')->boolean(),
        ];
    } // [tl! collapse:end]

    public function render(): View
    {
        return view('list-posts');
    }
}

```

Columns

Getting Started

Column classes can be found in the `Filament\Tables\Columns` namespace.

If you're using the columns in a Livewire component, you can put them in the `getTableColumns()` method:

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

If you're using them in admin panel resources or relation managers, you must put them in the `$table->columns()` method:

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

Columns may be created using the static `make()` method, passing its name. The name of the column should correspond to a column or accessor on your model. You may use "dot syntax" 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](#)
- [Badge column](#)
- [Tags 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 by using 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('title')->sortable()
```

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);
    })
```

If a column is `sortable()`, you may choose to sort it by default using the `getDefaultTableSortColumn()` and `getDefaultTableSortDirection()` methods:

```
protected function getDefaultTableSortColumn(): ?string
{
    return 'full_name';
}

protected function getDefaultTableSortDirection(): ?string
{
    return 'asc';
}
```

Persist sort in session

To persist the sort in the user's session, override the `shouldPersistTableSortInSession()` method on the Livewire component:

```
protected function shouldPersistTableSortInSession(): bool
{
    return true;
}
```

Searching

Columns may be searchable, by using the text input 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('title')->searchable()
```

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}%")
            ->where('last_name', 'like', "%{$search}%");
    })
```

Searching individually

You can choose to enable a per-column search input using the `isIndividual` parameter:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')->searchable(isIndividual: true)
```

If you use the `isIndividual` parameter, you may still search that column using the main "global" search input 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:

```
protected $queryString = [
    // ...
    'tableColumnSearchQueries',
];
```

Persist search in session

To persist the table or individual column search in the user's session, override the `shouldPersistTableSearchInSession()` or `shouldPersistTableColumnSearchInSession()` method on the Livewire component:

```
protected function shouldPersistTableSearchInSession(): bool
{
    return true;
}

protected function shouldPersistTableColumnSearchInSession(): bool
{
    return true;
}
```

Cell 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 behaviour of the action:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->action(function (Post $record): void {
        $this->dispatchBrowserEvent('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->dispatchBrowserEvent('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 fields with a `null` state, you may use the `default()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')->default('Untitled')
```

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('id')->toggleable()
```

By default, toggleable columns are visible. To make them hidden instead:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('id')->toggleable(isToggledHiddenByDefault: true)
```

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 `getStateUsing()` method, you can customize the returned state for that column based on the `$record`:

```
Tables\Columns\TextColumn::make('amount_including_vat')
->getStateUsing(function (Model $record): float {
    return $record->amount * (1 + $record->vat_rate);
})
```

Tooltips

If you want to use tooltips outside of the admin panel, make sure you have [@ryangjchandler/alpine-tooltip](#) installed in your app, including `tippy.css`. You'll also need to install `tippy.css` if you're using a [custom admin theme](#).

You may specify a tooltip to display when you hover over a cell:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
->tooltip('Title')
```

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}")
```

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 behaviour 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 `sortable()` and `toggleable()`, you can do it like so:

```
use Filament\Tables\Columns\Column;

Column::configureUsing(function (Column $column): void {
    $column
        ->toggleable()
        ->sortable();
});
```

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()
        ->sortable();
});
```

Of course, you are still able to overwrite this on each column individually:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('name')->toggleable(false)
```

Text

Text columns display simple text from your database:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
```

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)
```

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')
```

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()
```

Currency formatting

The `money()` method allows you to easily format monetary values, in any currency. This functionality uses [akaunting/laravel-money](#) internally:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('price')->money('eur')
```

Alternatively, you can set the default currency through the environment variable `DEFAULT_CURRENCY`.

Limiting text length

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->getLimit()) {
        return null;
    }

    // Only render the tooltip if the column contents exceeds the length limit.
    return $state;
})
```

Limiting word count

You may limit the number of `words()` displayed in the cell:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')->words(10)
```

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()
```

Rendering HTML

If your column value is HTML, you may render it using `html()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('description')->html()
```

Enum formatting

You may also transform a set of known cell values using the `enum()` method:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')->enum([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
```

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')->getStateUsing(
    static function (stdClass $rowLoop, HasTable $livewire): string {
        return (string) (
            $rowLoop->iteration +
            ($livewire->tableRecordsPerPage * (
                $livewire->page - 1
            ))
        );
    }
),
```

As `$rowLoop` is Laravel Blade's `$loop` object, you can reference all other `$loop` properties.

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}}"))
```

Adding a placeholder if the cell is empty

Sometimes you may want to display a placeholder if the cell's value is empty:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('updated_at')
    ->placeholder('Never')
```

Customizing the color

You may set a color for the text, either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('status')
    ->color('primary')
```

Adding an icon

Text columns may also have an icon:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->icon('heroicon-s-mail')
```

You may set the position of an icon using `iconPosition()`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->icon('heroicon-s-mail')
    ->iconPosition('after') // `before` or `after`
```

Customizing the text size

You may make the text smaller using `size('sm')`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
    ->size('sm')
```

Or you can make it larger using `size('lg')`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('title')
    ->size('lg')
```

Customizing the font weight

Text columns have regular font weight by default but you may change this to any of the the following options: `thin`, `extralight`, `light`, `medium`, `semibold`, `bold`, `extrabold` or `black`.

For instance, you may make the font bold using `weight('bold')`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('email')
->weight('bold')
```

Customizing the font family

You can change the text font family to any of the following options: `sans`, `serif` or `mono`.

For instance, you may make the font mono using `fontFamily('mono')`:

```
use Filament\Tables\Columns\TextColumn;

TextColumn::make('text')
->fontFamily('mono')
```

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)
```

Filament uses tooltips to display the copy message in the admin panel. If you want to use the copyable feature outside of the admin panel, make sure you have [@ryangjchandler/alpine-tooltip](#) installed in your app.

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}")
```

Icon

Icon columns render a Blade icon component representing their contents:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->options([
    'heroicon-o-x-circle',
    'heroicon-o-pencil' => 'draft',
    'heroicon-o-clock' => 'reviewing',
    'heroicon-o-check-circle' => 'published',
])
```

You may also pass a callback to activate an option, accepting the cell's `$state` and `$record`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->options([
    'heroicon-o-x-circle',
    'heroicon-o-pencil' => fn ($state, $record): bool => $record->status === 2,
    'heroicon-o-clock' => fn ($state): bool => $state === 'reviewing',
    'heroicon-o-check-circle' => fn ($state): bool => $state === 'published',
])
```

Customizing the color

Icon columns may also have a set of icon colors, using the same syntax. They may be either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
->options([
    'heroicon-o-x-circle',
    'heroicon-o-pencil' => 'draft',
    'heroicon-o-clock' => 'reviewing',
    'heroicon-o-check-circle' => 'published',
])
->colors([
    'secondary',
    'danger' => 'draft',
    'warning' => 'reviewing',
    'success' => 'published',
])
```

Customizing the size

The default icon size is `lg`, but you may customize the size to be either `xs`, `sm`, `md`, `lg` or `xl`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
    ->options([
        'heroicon-s-x-circle',
        'heroicon-s-pencil' => 'draft',
        'heroicon-s-clock' => 'reviewing',
        'heroicon-s-check-circle' => 'published',
    ])
    ->size('md')
```

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()
```

Customizing the boolean icons

You may customize the icon representing each state. Icons are the name of a Blade component present. By default, [Heroicons v1](#) are installed:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
    ->boolean()
    ->trueIcon('heroicon-o-badge-check')
    ->falseIcon('heroicon-o-x-circle')
```

Customizing the boolean colors

You may customize the icon color representing each state. These may be either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Tables\Columns\IconColumn;

IconColumn::make('is_featured')
    ->boolean()
    ->trueColor('primary')
    ->falseColor('warning')
```

Image

Images can be easily displayed within your table:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('header_image')
```

The column in the database must contain the path to the image, relative to the root directory of its storage disk.

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')
```

Square image

You may display the image using a 1:1 aspect ratio:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('author.avatar')->square()
```

Circular image

You may make the image fully rounded, which is useful for rendering avatars:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('author.avatar')->circular()
```

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)
```

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'))
```

Custom attributes

You may customize the extra HTML attributes of the image using `extraImgAttributes()`:

```
use Filament\Tables\Columns\ImageColumn;

ImageColumn::make('logo')
    ->extraImgAttributes(['title' => 'Company logo']),
```

Badge

Badge columns render a colored badge with the cell's contents:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
->enum([
    'draft' => 'Draft',
    'reviewing' => 'Reviewing',
    'published' => 'Published',
])
```

Customizing the color

Badges may have a color. It may be either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
->colors([
    'primary',
    'secondary' => 'draft',
    'warning' => 'reviewing',
    'success' => 'published',
    'danger' => 'rejected',
])
```

You may instead activate a color using a callback, accepting the cell's `$state`:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
->colors([
    'primary',
    'secondary' => static fn ($state): bool => $state === 'draft',
    'warning' => static fn ($state): bool => $state === 'reviewing',
    'success' => static fn ($state): bool => $state === 'published',
    'danger' => static fn ($state): bool => $state === 'rejected',
])
```

Or dynamically calculate the color based on the `$record` and / or `$state`:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
    ->color(static function ($state): string {
        if ($state === 'published') {
            return 'success';
        }

        return 'secondary';
    })
)
```

Adding an icon

Badges may also have an icon:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
    ->icons([
        'heroicon-o-x',
        'heroicon-o-document' => 'draft',
        'heroicon-o-refresh' => 'reviewing',
        'heroicon-o-truck' => 'published',
    ])
)
```

Alternatively, you may conditionally display an icon using a closure:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
    ->icons([
        'heroicon-o-x',
        'heroicon-o-document' => static fn ($state): bool => $state === 'draft',
        'heroicon-o-refresh' => static fn ($state): bool => $state === 'reviewing',
        'heroicon-o-truck' => static fn ($state): bool => $state === 'published',
    ])
)
```

Or dynamically calculate the icon based on the `$record` and / or `$state`:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
    ->icon(static function ($state): string {
        if ($state === 'published') {
            return 'heroicon-o-truck';
        }

        return 'heroicon-o-x';
    })
)
```

You may set the position of an icon using `iconPosition()`:

```
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('status')
    ->icons([
        'heroicon-o-x',
        'heroicon-o-document' => 'draft',
        'heroicon-o-refresh' => 'reviewing',
        'heroicon-o-truck' => 'published',
    ])
    ->iconPosition('after') // `before` or `after`
```

Formatting the text

All formatting options available for [text columns](#) are also available for badge columns.

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\BadgeColumn;

BadgeColumn::make('email')
    ->copyable()
    ->copyMessage('Email address copied')
    ->copyMessageDuration(1500)
```

Filament uses tooltips to display the copy message in the admin panel. If you want to use the copyable feature outside of the admin panel, make sure you have [@ryangjchandler/alpine-tooltip](#) installed in your app.

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\BadgeColumn;

BadgeColumn::make('name')
    ->copyable()
    ->copyableState(fn (string $state): string => "Name: {$state}")
```

In this function, you can access the whole table row with `$record`:

```
use App\Models\User;
use Filament\Tables\Columns\BadgeColumn;

BadgeColumn::make('name')
    ->copyable()
    ->copyableState(fn (User $record): string => "Name: {$record->name}")
```

Tags

Tags columns render a list of tags from an array:

```
use Filament\Tables\Columns\TagsColumn;

TagsColumn::make('tags')
```

Be sure to add an `array` cast to the model property:

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

class Post extends Model
{
    protected $casts = [
        'tags' => 'array',
    ];

    // ...
}
```

Using a separator

Instead of using an array, you may use a separated string by passing the separator into `separator()`:

```
use Filament\Tables\Columns\TagsColumn;

TagsColumn::make('tags')->separator(',')
```

Color

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')
```

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)
```

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}")
```

Select

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',
    ])
```

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 `disablePlaceholderSelection()` method:

```
use Filament\Tables\Columns\SelectColumn;

SelectColumn::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
    ->disablePlaceholderSelection()
```

Toggle

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')
```

Text Input

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('name')
```

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'])
```

Filament uses tooltips to display validation errors. If you want to use tooltips outside of the admin panel to display validation errors, make sure you have [@ryangjchandler/alpine-tooltip](#) installed in your app.

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')
```

Checkbox

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')
```

Custom

View column

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')
```

Inside your view, you may retrieve the state of the cell using the `$getState()` method:

```
<div>
{{ $getState() }}
</div>
```

You can also access the entire Eloquent record with `$getRecord()`.

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';
}
```

Inside your view, you may retrieve the state of the cell using the `$getState()` method:

```
<div>
{{ $getState() }}
</div>
```

You can also access the entire Eloquent record with `$getRecord()`.

Relationships

Displaying data from relationships

You may use "dot syntax" 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.

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.

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.

Filters

Getting started

Filters allow you to scope the Eloquent query as a way to reduce the number of records in a table.

If you're using the filters in a Livewire component, you can put them in the `getTableFilters()` method:

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

If you're using them in admin panel resources or relation managers, you must put them in the `$table->filters()` method:

```
public static function table(Table $table): Table
{
    return $table
        ->filters([
            // ...
        ]);
}
```

Filters may be created using the static `make()` method, passing its name. The name of the filter should be unique. 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))
```

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 by using the `translateLabel()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')->translateLabel() // Equivalent to `label(__('Is featured'))`
```

Using a toggle button instead of a checkbox

By default, filters use a checkbox to control the filter. Instead, you may switch to using a toggle button, using the `toggle()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')->toggle()
```

Default filters

You may set a filter to be enabled by default, using the `default()` method:

```
use Filament\Tables\Filters\Filter;

Filter::make('is_featured')->label('Featured')->default()
```

Filter forms

By default, filters have two states: enabled and disabled. When the filter is enabled, it is applied to the query. When it is disabled it is not. This is controlled through a checkbox. However, some filters may require extra data input to narrow down the results further. You may use a custom filter form to collect this data.

Select filters

Select filters allow you to quickly create a filter that allows the user to select an option 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:

```
use Filament\Tables\Filters\SelectFilter;

SelectFilter::make('status')
    ->options([
        'draft' => 'Draft',
        'reviewing' => 'Reviewing',
        'published' => 'Published',
    ])
```

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:

```
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 `BelongsTo` 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')
```

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())
```

Ternary filters

Ternary filters allow you to quickly create a 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')
```

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()
```

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')
```

You may customize the query used for each state of the ternary filter, using the `queries()` method:

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

TernaryFilter::make('trashed')
->placeholder('Without trashed records')
->trueLabel('With trashed records')
->falseLabel('Only trashed records')
->queries(
    true: fn (Builder $query) => $query->withTrashed(),
    false: fn (Builder $query) => $query->onlyTrashed(),
    blank: fn (Builder $query) => $query->withoutTrashed(),
)
```

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;
use Filament\Tables\Filters\Filter;
use Illuminate\Database\Eloquent\Builder;

Filter::make('created_at')
->form([
    Forms\Components\DatePicker::make('created_from'),
    Forms\Components\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),
        );
})
```

Setting default values

If you wish to set a default filter value, you may use the `default()` method on the form component:

```
use Filament\Forms;
use Filament\Tables\Filters\Filter;
use Illuminate\Database\Eloquent\Builder;

Filter::make('created_at')
->form([
    Forms\Components\DatePicker::make('created_from'),
    Forms\Components\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.

By default, the label of the filter is used as the indicator. You can override this:

```
use Filament\Tables\Filters\TernaryFilter;

TernaryFilter::make('is_admin')
->label('Administrators only?')
->indicator('Administrators')
```

Custom 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 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();
})
```

You may even render multiple indicators at once, by returning an array. If you have different fields associated with different indicators, you should use the field's name as the array key, to ensure that the correct field is reset when the filter is removed:

```

use Filament\Forms\Components\DatePicker;
use Filament\Tables\Filters\Filter;

Filter::make('created_at')
->form([
    DatePicker::make('from'),
    DatePicker::make('until'),
])
// ...
->indicateUsing(function (array $data): array {
    $indicators = [];

    if ($data['from'] ?? null) {
        $indicators['from'] = 'Created from ' . Carbon::parse($data['from'])-
>toFormattedDateString();
    }

    if ($data['until'] ?? null) {
        $indicators['until'] = 'Created until ' . Carbon::parse($data['until'])-
>toFormattedDateString();
    }

    return $indicators;
})

```

Apearance

By default, filters are displayed in a thin popover on the right side of the table, in 1 column.

To change the number of columns that filters may occupy, you may use the `getTableFiltersFormColumns()` method:

```

protected function getTableFiltersFormColumns(): int
{
    return 3;
}

```

Adding more columns to the filter form will automatically widen the popover. To customize the popover width, you may use the `getTableFiltersFormWidth()` method, and specify a width from `xs` to `7xl`:

```

protected function getTableFiltersFormWidth(): string
{
    return '4xl';
}

```

Displaying filters above or below the table content

To render the filters above the table content instead of in a popover, you may use:

```
use Filament\Tables\Filters\Layout;

protected function getTableFiltersLayout(): ?string
{
    return Layout::AboveContent;
}
```

To render the filters below the table content instead of in a popover, you may use:

```
use Filament\Tables\Filters\Layout;

protected function getTableFiltersLayout(): ?string
{
    return Layout::BelowContent;
}
```

Persist filters in session

To persist the table filters in the user's session, use the `shouldPersistTableFiltersInSession()` method:

```
protected function shouldPersistTableFiltersInSession(): bool
{
    return true;
}
```

Actions

Getting started

Single actions

Single action buttons are rendered at the end of each table row.

If you're using the actions in a Livewire component, you can put them in the `getTableActions()` method.

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

If you're using them in admin panel resources or relation managers, you must put them on the `$table`:

```
public static function table(Table $table): Table
{
    return $table
        ->actions([
            // ...
        ]);
}
```

Actions may be created using the static `make()` method, passing its name. The name of the action should be unique. You can then pass a callback to `action()` which executes the task, or a callback to `url()` which generates a link URL:

If you would like the URL to open in a new tab, you can use the `openUrlInNewTab()` method.

```
use App\Models\Post;
use Filament\Tables\Actions\Action;

Action::make('edit')
    ->url(fn (Post $record): string => route('posts.edit', $record))
    ->openUrlInNewTab()
```

Bulk actions

Bulk action buttons are visible when the user selects at least one record.

If you're using the actions in a Livewire component, you can put them in the `getTableBulkActions()` method.

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

If you're using them in admin panel resources or relation managers, you must put them on the `$table`:

```
public static function table(Table $table): Table
{
    return $table
        ->bulkActions([
            // ...
        ]);
}
```

Bulk actions may be created using the static `make()` method, passing its name. The name of the action should be unique. 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')
    ->action(fn (Collection $records) => $records->each->delete())
```

Please note that Filament uses the parameter name `$records` in the function to inject the collection. Any other parameter name will resolve from the container instead.

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()
```

Setting a label

By default, the label of the action is generated from its name. You may customize this using the `label()` method:

```
use App\Models\Post;
use Filament\Tables\Actions\Action;

Action::make('edit')
    ->label('Edit post')
    ->url(fn (Post $record): string => route('posts.edit', $record))
```

Optionally, you can have the label automatically translated by using the `translateLabel()` method:

```
use App\Models\Post;
use Filament\Tables\Actions\Action;

Action::make('edit')
    ->translateLabel() // Equivalent to `label(__('Edit'))`
    ->url(fn (Post $record): string => route('posts.edit', $record))
```

Setting a color

Actions may have a color to indicate their significance. It may be either `primary`, `secondary`, `success`, `warning` or `danger`:

```
use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->action(fn (Collection $records) => $records->each->delete())
    ->deselectRecordsAfterCompletion()
    ->color('danger')
```

Disabling record bulk actions

You may conditionally disable bulk actions for a specific record:

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

public function isTableRecordSelectable(): ?Closure
{
    return fn (Model $record): bool => $record->status === Status::Enabled;
}
```

Setting a size

The default size for table actions is `sm` but you may also change it to either `md` or `lg`:

```
use Filament\Tables\Actions\Action;

Action::make('delete')
    ->size('lg')
```

Setting an icon

Bulk actions and some single actions may also render a Blade icon component to indicate their purpose:

```

use App\Models\Post;
use Filament\Tables\Actions\Action;
use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->action(fn (Collection $records) => $records->each->delete())
    ->deselectRecordsAfterCompletion()
    ->color('danger')
    ->icon('heroicon-o-trash')

Action::make('edit')
    ->label('Edit post')
    ->url(fn (Post $record): string => route('posts.edit', $record))
    ->icon('heroicon-s-pencil')

```

Modals

Actions and bulk actions may require additional confirmation or form information before they run. With the table builder, 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 Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->action(fn (Collection $records) => $records->each->delete())
    ->deselectRecordsAfterCompletion()
    ->requiresConfirmation()

```

Note: 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()` callback.

Custom forms

You may also render a form in this 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()` callback:

```

use App\Models\User;
use Filament\Forms;
use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('updateAuthor')
    ->action(function (Collection $records, array $data): void {
        foreach ($records as $record) {
            $record->author()->associate($data['authorId']);
            $record->save();
        }
    })
    ->form([
        Forms\Components\Select::make('authorId')
            ->label('Author')
            ->options(User::query()->pluck('name', 'id'))
            ->required(),
    ])
)

```

Filling default data

You may fill the form with default data, using the `[mountUsing()]` method:

```

use App\Models\User;
use Filament\Forms;
use Filament\Tables\Actions\Action;
use Illuminate\Database\Eloquent\Collection;

Action::make('updateAuthor')
    ->mountUsing(fn (Forms\ComponentContainer $form, User $record) => $form->fill([
        'authorId' => $record->author->id,
    ]))
    ->action(function (User $record, array $data): void {
        $record->author()->associate($data['authorId']);
        $record->save();
    })
    ->form([
        Forms\Components\Select::make('authorId')
            ->label('Author')
            ->options(User::query()->pluck('name', 'id'))
            ->required(),
    ])
)

```

Wizards

You may easily transform action forms into multistep wizards.

On the action, simply pass in the wizard steps to the `[steps()]` method, instead of `[form()]`:

```

use Filament\Forms\Components\MarkdownEditor;
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Toggle;
use Filament\Forms\Components\Wizard\Step;
use Filament\Tables\Actions\Action;

Action::make('create')
    ->steps([
        Step::make('Name')
            ->description('Give the category a clear and unique name')
            ->schema([
                TextInput::make('name')
                    ->required()
                    ->reactive()
                    ->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),
            ]),
    ])
)

```

Setting a modal heading, subheading, and button label

You may customize the heading, subheading and button label of the modal:

```

use Filament\Tables\Actions\BulkAction;
use Illuminate\Database\Eloquent\Collection;

BulkAction::make('delete')
    ->action(fn (Collection $records) => $records->each->delete())
    ->deselectRecordsAfterCompletion()
    ->requiresConfirmation()
    ->modalHeading('Delete posts')
    ->modalSubheading('Are you sure you\'d like to delete these posts? This cannot be undone.')
    ->modalButton('Yes, delete them')

```

Custom 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 Filament\Tables\Actions\BulkAction;

BulkAction::make('advance')
    ->action(fn () => $this->record->advance())
    ->modalContent(view('filament.resources.event.actions.advance'))
```

By default, the custom content is displayed above the modal form if there is one, but you can add content below using `modalFooter()` if you wish:

```
use Filament\Pages\Actions\BulkAction;

BulkAction::make('advance')
    ->action(fn () => $this->record->advance())
    ->modalFooter(view('filament.resources.event.actions.advance'))
```

Authorization

You may conditionally show or hide actions and bulk actions for certain users using either the `visible()` or `hidden()` methods, passing a closure:

```
use App\Models\Post;
use Filament\Tables\Actions\Action;

Action::make('edit')
    ->url(fn (Post $record): string => route('posts.edit', $record))
    ->visible(fn (Post $record): bool => auth()->user()->can('update', $record))
```

This is useful for authorization of certain actions to only users who have permission.

Prebuilt actions

Replicate

This package includes an action to replicate table records. You may use it like so:

```
use Filament\Tables\Actions\ReplicateAction;

ReplicateAction::make()
```

The `excludeAttributes()` method is used to instruct the action which columns to be excluded from replication:

```
use Filament\Tables\Actions\ReplicateAction;

ReplicateAction::make()->excludeAttributes(['slug'])
```

The `beforeReplicaSaved()` method can be used to invoke a Closure before saving the replica:

```
use Filament\Tables\Actions\ReplicateAction;
use Illuminate\Database\Eloquent\Model;

ReplicateAction::make()
    ->beforeReplicaSaved(function (Model $replica): void {
        // ...
    })
}
```

The `afterReplicaSaved()` method can be used to invoke a Closure after saving the replica:

```
use Filament\Tables\Actions\ReplicateAction;
use Illuminate\Database\Eloquent\Model;

ReplicateAction::make()
    ->afterReplicaSaved(function (Model $replica): void {
        // ...
    })
}
```

Retrieving user input

Just like [normal actions](#), you can provide a [form schema](#) that can be used to modify the replication process:

```
use Filament\Tables\Actions\ReplicateAction;

ReplicateAction::make()
    ->excludeAttributes(['title'])
    ->form([
        TextInput::make('title')->required(),
    ])
    ->beforeReplicaSaved(function (Model $replica, array $data): void {
        $replica->fill($data);
    })
)
```

Grouping

You may use an `ActionGroup` object to group multiple table actions together in a dropdown:

```
use Filament\Tables;

protected function getTableActions(): array
{
    return [
        Tables\Actions\ActionGroup::make([
            Tables\Actions\ViewAction::make(),
            Tables\Actions>EditAction::make(),
            Tables\Actions>DeleteAction::make(),
        ]),
    ];
}
```

Position

By default, the row actions in your table are rendered in the final cell. You may change the position by overriding the `getTableActionsPosition()` method:

```
use Filament\Tables\Actions\Position;

protected function getTableActionsPosition(): ?string
{
    return Position::BeforeCells;
}
```

Alignment

Row actions are aligned to the right in their cell by default. To change the alignment, update the configuration value inside of the package config:

```
'actions' => [
    'cell' => [
        'alignment' => 'right', // `right`, `left` or `center`
    ],
]
```

Tooltips

If you want to use tooltips outside of the admin panel, make sure you have [@ryangjchandler/alpine-tooltip installed](#) in your app, including `tippy.css`. You'll also need to install `tippy.css` if you're using a [custom admin theme](#).

You may specify a tooltip to display when you hover over an action:

```
use Filament\Tables\Actions\Action;

Action::make('edit')
->tooltip('Edit this blog post')
```

This method also accepts a closure that can access the current table record:

```
use Filament\Tables\Actions\Action;
use Filament\Tables\Columns\TextColumn;
use Illuminate\Database\Eloquent\Model;

Action::make('edit')
->tooltip(fn (Model $record): string => "Edit {$record->title}")
```

Layout

The problem with traditional table layouts

Traditional tables are notorious for having bad responsivity. 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.

Sort by	Search
Scotty Rempel III Manager of Air Crew	+16157401014 wkirlin@example.net
Eloisa Quitzon Computer Security Specialist	1-248-890-2049 dubuque.claudine@example.com
Mr. Alvah Stiedemann III Music Composer	+17478074427 erika12@example.org
Jolie DuBuque Jr. Glass Blower	+1(740) 592-3567 jalen.stokes@example.com
Prof. Mathilde Cremin Cutting Machine Operator	+1(678) 340-8025 xstreich@example.org
Rafaela Leffler Food Servers	1-978-398-3548 bennie23@example.net
Miss Fanny Schowalter MD Geologist	+1(313) 578-7341 trantow.abigayle@example.co
Dr. Kay Kling Sr.	+1386.815.8420

Sort by	Search
Scotty Rempel III Manager of Air Crew	Edit
Eloisa Quitzon Computer Security Specialist	Edit
Mr. Alvah Stiedemann III Music Composer	Edit
Jolie DuBuque Jr. Glass Blower	Edit
Prof. Mathilde Cremin Cutting Machine Operator	Edit

Allowing columns to stack on mobile

Let's introduce a component - `Split`:

```
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar'),
    TextColumn::make('name'),
    TextColumn::make('email'),
])
])
```

Avatar	Name	Email
	Scotty Rempel III	wkirlin@example.net
	Eloisa Quitzon	dubuque.claudine@example.com
	Mr. Alvah Stiedemann III	erika12@example.org
	Jolie DuBuque Jr.	jalen.stokes@example.com
	Prof. Mathilde Cremin	xstreich@example.org
	Rafaela Leffler	bennie23@example.com
	Miss Fanny Schowalter MD	trantow.a@example.org
	Dr. Kay Kling Sr.	ashton.jas@example.com

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 behaviour starts `from()`. Before this point, the columns will stack on top of each other:

```
use Filament\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar'),
    TextColumn::make('name'),
    TextColumn::make('email'),
])->from('md')
```

In this example, the columns will only appear horizontally aside each other from `md` `breakpoint` devices onwards:

	Scotty Rempel III	wkirlin@example.net
<input type="checkbox"/>	Eloisa Quitzon	dubuque.claudine@example.com
<input type="checkbox"/>	Mr. Alvah Stiedemann III	erika12@example.org
<input type="checkbox"/>	Jolie DuBuque Jr.	jalen.stokely@example.org
<input type="checkbox"/>	Prof. Mathilde Cremin	xstreich@example.org
<input type="checkbox"/>	Rafaela Leffler	bennie23@example.org
<input type="checkbox"/>	Miss Fanny Schowalter MD	trantow.audrey@example.org
<input type="checkbox"/>	Dr. Kay Kling Sr.	ashton.jace@example.org

	Scotty Rempel III	wkirlin@example.net
<input type="checkbox"/>	Eloisa Quitzon	dubuque.claudine@example.com
<input type="checkbox"/>	Mr. Alvah Stiedemann III	erika12@example.org

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\Tables\Columns\Layout\Split;
use Filament\Tables\Columns\ImageColumn;
use Filament\Tables\Columns\TextColumn;

Split::make([
    ImageColumn::make('avatar')->grow(false),
    TextColumn::make('name'),
    TextColumn::make('email'),
])
```

The other columns which are allowed to `grow()` will adjust to consume the newly-freed space:

The screenshot shows a table component on the left and a modal dialog on the right. The table has columns for 'Sort by' (dropdown), 'Avatar' (image), 'Name' (text), and 'Email' (text). The modal dialog also has a 'Sort by' dropdown, an 'Edit' button, and a 'Delete' button.

Sort by	Avatar	Name	Email
<input type="checkbox"/>		Scotty Rempel III	wkirlin@example.net
<input type="checkbox"/>		Eloisa Quitzon	dubuque.claudine@example.com
<input type="checkbox"/>		Mr. Alvah Stiedemann III	erikal2@example.org
<input type="checkbox"/>		Jolie DuBuque Jr.	jalen.stokes@example.com
<input type="checkbox"/>		Prof. Mathilde Cremin	xstreich@example.org
<input type="checkbox"/>		Rafaela Leffler	bennie23@example.net
<input type="checkbox"/>		Miss Fanny Schowalter MD	trantow.abigayle@example.com
<input type="checkbox"/>		Dr. Kay Kling Sr.	ashton.jast@example.org

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\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'),
    TextColumn::make('name'),
    Stack::make([
        TextColumn::make('email'),
        TextColumn::make('phone'),
    ]),
])
```

<input type="checkbox"/>	Sort by	-	<input type="button" value="▼"/>
<input type="checkbox"/>	 Scotty Rempel III	 +16157401014  wkirlin@example.net	<input type="button" value="Search"/> <input type="button" value="▼"/>
<input type="checkbox"/>	 Eloisa Quitzon	 1-248-890-2049  dubuque.claudine@example.com	<input type="button" value="Search"/> <input type="button" value="▼"/>
<input type="checkbox"/>	 Mr. Alvah Stiedemann III	 +17478074427  erikal2@example.org	<input type="checkbox"/> Sort by - <input type="button" value="▼"/>
<input type="checkbox"/>	 Jolie DuBuque Jr.	 +1(740) 592-3567  jalen.stokes@example.com	<input type="button" value="Edit"/> <input type="checkbox"/>
<input type="checkbox"/>	 Prof. Mathilde Cremin	 +(678) 340-8025  xstreich@example.org	<input type="button" value="Edit"/> <input type="checkbox"/>
<input type="checkbox"/>	 Rafaela Leffler	 1-978-398-3548  bennie23@example.net	<input type="button" value="Edit"/> <input type="checkbox"/>
<input type="checkbox"/>	 Miss Fanny Schowalter MD	 +(313) 578-7341  trantow.abigayle@example.com	<input type="button" value="Edit"/> <input type="checkbox"/>
<input type="checkbox"/>	 Dr. Kay Kling Sr.	 +1.386.815.8420  ashton.jast@example.org	<input type="checkbox"/> Mr. Alvah Stiedemann III

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\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'),
    TextColumn::make('name'),
    Stack::make([
        TextColumn::make('email'),
        TextColumn::make('phone'),
    ]) ->visibleFrom('md'),
])
```

<input type="checkbox"/>	Sort by - ▾	<input type="text"/> Search
<input type="checkbox"/>	Scotty Rempel III	+16157401014 wkirlin@example.net
<input type="checkbox"/>	Eloisa Quitzon	1-248-890-2049 dubuque.claudine@example.com
<input type="checkbox"/>	Mr. Alvah Stiedemann III	+17478074427 erikal2@example.org
<input type="checkbox"/>	Jolie DuBuque Jr.	+1(740) 592-3567 jalen.stokes@example.com
<input type="checkbox"/>	Prof. Mathilde Cremin	+(678) 340-8025 xstreich@example.org
<input type="checkbox"/>	Rafaela Leffler	1-978-398-3548 bennie23@example.net
<input type="checkbox"/>	Miss Fanny Schowalter MD	+(313) 578-7341 trantow.abigayle@example.com
<input type="checkbox"/>	Dr. Kay Kling Sr.	+1.386.815.8420 ashton.jast@example.org

Aligning stacked content

By default, columns within a stack are aligned to the left. You may choose to align columns within a stack to the `center` or `right`:

```
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'),
    TextColumn::make('name'),
    Stack::make([
        TextColumn::make('email'),
        TextColumn::make('phone'),
    ]) ->alignment('right'),
])
```

<input type="checkbox"/>		Scotty Rempel III	+16157401014	Edit
<input type="checkbox"/>		Eloisa Quitzon	1-248-890-2049	Edit
<input type="checkbox"/>		Mr. Alvah Stiedemann III	+17478074427	Edit
<input type="checkbox"/>		Jolie DuBuque Jr.	+1 (740) 592-3567	Edit
<input type="checkbox"/>		Prof. Mathilde Cremin	+1 (678) 340-8025	Edit
<input type="checkbox"/>		Rafaela Leffler	1-978-398-3548	Edit
<input type="checkbox"/>		Miss Fanny Schowalter MD	+1 (313) 578-7341	Edit
<input type="checkbox"/>		Dr. Kay Kling Sr.	+1.386.815.8420	Edit

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('email'),
    TextColumn::make('phone'),
])->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('email'),
    TextColumn::make('phone'),
])
```

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('email'),
    TextColumn::make('phone'),
])
])
```

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('email')
        ->columnSpan([
            '2xl' => 2,
        ]),
    TextColumn::make('phone'),
])
])
```

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\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'),
        TextColumn::make('name'),
    ]),
    Panel::make([
        Stack::make([
            TextColumn::make('email'),
            TextColumn::make('phone'),
        ]),
    ]) ->collapsible(),
]

```

You can expand a panel by default using the `collapsed(false)` method:

```

use Filament\Tables\Columns\Layout\Panel;
use Filament\Tables\Columns\TextColumn;

Panel::make([
    Stack::make([
        TextColumn::make('email'),
        TextColumn::make('phone'),
    ]),
]) ->collapsed(false)

```

Adding a collapse animation

If you're not using the table builder within the admin panel, you may find that there is no animation when collapsing or expanding the content. You can enable this by installing the [Alpine.js Collapse Plugin](#):

```
npm install @alpinejs/collapse --save-dev
```

Finally, import `@alpinejs/collapse` as an Alpine.js plugin in your JavaScript file:

```
import Alpine from 'alpinejs'
import Collapse from '@alpinejs/collapse'

Alpine.plugin(Collapse)
```

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 define a new `getTableContentGrid()` method on your Livewire component:

```
protected function getTableContentGrid(): ?array
{
    return [
        'md' => 2,
        'xl' => 3,
    ];
}
```

Or if you're using admin panel resources or relation managers, you must define a `$table->contentGrid()` method:

```
public static function table(Table $table): Table
{
    return $table
        ->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.

Custom HTML

You may add custom HTML to your table using a `View` component. It can even be `collapsible()`:

```
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'),
        TextColumn::make('name'),
    ]),
    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\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'),
        TextColumn::make('name'),
    ]),
    View::make('users.table.collapsible-row-content')
        ->components([
            TextColumn::make('email'),
        ])
        ->collapsible(),
]
]
```

Now, render the components in the Blade file:

```
<div class="px-4 py-3 bg-gray-100 rounded-lg">
    <x-tables::columns.layout
        :components="$getComponents()"
        :record="$getRecord()"
        :record-key="$recordKey"
    />
</div>
```

Testing

All examples in this guide will be written using [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);
});
```

Note that if your table uses pagination, `assertCanSeeTableRecords()` will only check for records on the first page. To switch page, call `set('page', 2)`.

Note that 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 `assertCanRenderTableColumn()`:

```
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`);
});
```

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 `assertSelectColumnHasOptions()` and `assertSelectColumnDoesNotHaveOptions()`:

```
use function Pest\Livewire\livewire;

it('has the correct statuses', function () {
    $post = Post::factory()->create();

    livewire(PostsTable::class)
        ->assertSelectColumnHasOptions('status', ['unpublished' => 'Unpublished', 'published' => 'Published'], $post)
        ->assertSelectColumnDoesNotHaveOptions('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-o-archive-box')
        ->assertTableActionDoesNotHaveIcon('delete', 'heroicon-o-trash');
        ->assertTableBulkActionHasIcon('delete', 'heroicon-o-archive-box')
        ->assertTableBulkActionDoesNotHaveIcon('delete', 'heroicon-o-trash');

});
});
```

To ensure 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');  
});
```

Chapter 4

Notifications

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

Notifications come pre-installed inside the [admin panel 2.x](#), but you must still follow the installation instructions below if you're using it in the rest of your app.

First, require the notifications package using Composer:

```
composer require filament/notifications:"^2.0"
```

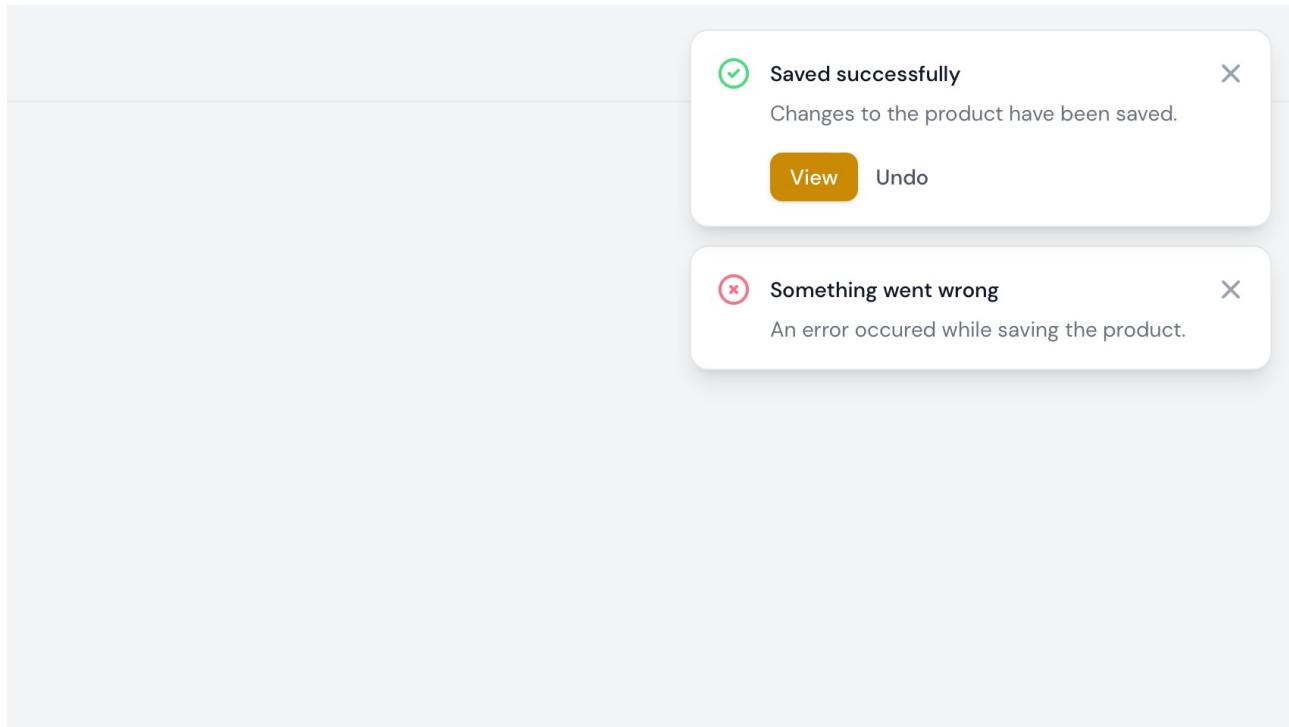
New Laravel projects

To get started with notifications quickly, you can set up [Livewire](#), [Alpine.js](#) and [Tailwind CSS](#) with these commands:

```
php artisan notifications:install  
npm install  
npm run dev
```

These commands will ruthlessly overwrite existing files in your application, hence why we only recommend using this method for new projects.

You're now ready to start [sending notifications!](#)



Existing Laravel projects

The package uses the following dependencies:

- [Alpine.js](#)
- [Alpine Floating UI](#)
- [PostCSS](#)
- [Tailwind CSS](#)

You may install these through NPM:

```
npm install alpinejs @awcodes/alpine-floating-ui postcss tailwindcss --save-dev
```

Configuring Tailwind CSS

To finish installing Tailwind, you must create a new `tailwind.config.js` file in the root of your project. The easiest way to do this is by running `npx tailwindcss init`.

In `tailwind.config.js`, add custom colors used by notifications:

```
import colors from 'tailwindcss/colors' // [tl! focus]

export default {
    content: [
        './resources/**/*.{blade.php}',
        './vendor/filament/**/*.{blade.php}', // [tl! focus]
    ],
    theme: {
        extend: {
            colors: { // [tl! focus:start]
                danger: colors.rose,
                primary: colors.blue,
                success: colors.green,
                warning: colors.yellow,
            }, // [tl! focus:end]
        },
    },
}
```

Of course, you may specify your own custom `primary`, `success`, `warning` and `danger` colors, which will be used instead.

Bundling assets

New Laravel projects use Vite for bundling assets by default. However, your project may still use Laravel Mix. Read the steps below for the bundler used in your project.

Vite

If you're using Vite, you should manually install [Autoprefixer](#) through NPM:

```
npm install autoprefixer --save-dev
```

Create a `postcss.config.js` file in the root of your project, and register Tailwind CSS and Autoprefixer as plugins:

```
export default {
    plugins: {
        tailwindcss: {},
        autoprefixer: {},
    },
}
```

You may also want to update your `vite.config.js` file to refresh the page after Livewire components have been updated:

```
import { defineConfig } from 'vite';
import laravel, { refreshPaths } from 'laravel-vite-plugin'; // [tl! focus]

export default defineConfig({
    plugins: [
        laravel({
            input: [
                'resources/css/app.css',
                'resources/js/app.js',
            ],
            refresh: [ // [tl! focus:start]
                ...refreshPaths,
                'app/Http/Livewire/**',
            ], // [tl! focus:end]
        }),
    ],
});
```

Laravel Mix

In your `webpack.mix.js` file, register Tailwind CSS as a PostCSS plugin:

```
const mix = require('laravel-mix')

mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css', [
        require('tailwindcss'), // [tl! focus]
    ])
])
```

Configuring styles

In `/resources/css/app.css`, import Tailwind CSS:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Configuring scripts

In `/resources/js/app.js`, import Alpine.js, Alpine Floating UI and the `filament/notifications` plugin, and register them:

```

import Alpine from 'alpinejs'
import AlpineFloatingUI from '@awcodes/alpine-floating-ui'
import NotificationsAlpinePlugin from '../../../../../vendor/filament/notifications/dist/module.esm'

Alpine.plugin(AlpineFloatingUI)
Alpine.plugin(NotificationsAlpinePlugin)

window.Alpine = Alpine

Alpine.start()

```

Compiling assets

Compile your new CSS and JS assets using `npm run dev`.

Configuring layout

Finally, create a new `resources/views/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>
        @vite(['resources/css/app.css', 'resources/js/app.js'])
        @livewireStyles
        @livewireScripts
        @stack('scripts')
    </head>

    <body class="antialiased">
        {{ $slot }}

        @livewire('notifications')
    </body>
</html>

```

You're now ready to start [sending notifications!](#)

Publishing configuration

If you wish, you may publish the configuration of the package using:

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

Upgrading

To upgrade the package to the latest version, you must run:

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

We recommend adding the `filament:upgrade` command to your `composer.json`'s `post-update-cmd` to run it automatically:

```
"post-update-cmd": [  
    // ...  
    "@php artisan filament:upgrade"  
,
```

Sending Notifications

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\Http\Livewire;

use Filament\Notifications\Notification; // [tl! focus]
use Livewire\Component;

class EditPost extends Component
{
    public function save(): void
    {
        // ...

        Notification::make() // [tl! focus:start]
            ->title('Saved successfully')
            ->success()
            ->send(); // [tl! focus:end]
    }
}
```



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') // [tl! focus]
    ->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully') // [tl! focus]
    .send()
```

Markdown text will automatically be rendered if passed to the title.

Icon

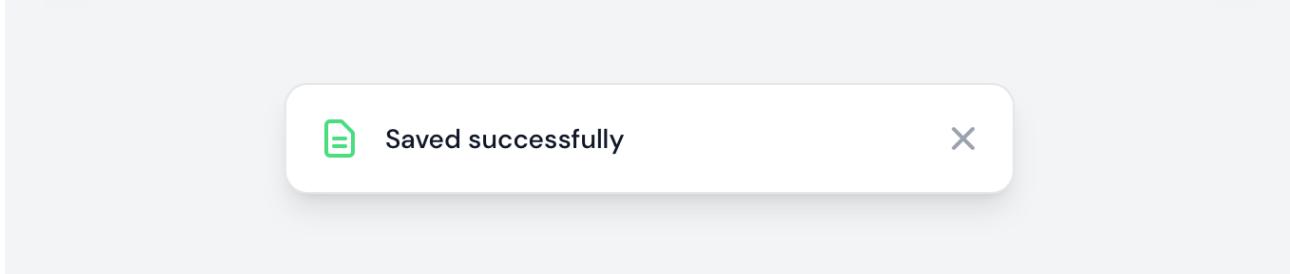
Optionally, a notification can have an icon that's displayed in front of its content. You may also set a color for the icon, which defaults to the `secondary` color specified in your `tailwind.config.js` file. The icon can be the name of any Blade component. By default, the [Blade Heroicons v1](#) package is installed, so you may use the name of any [Heroicons v1](#) out of the box. However, you may create your own custom icon components or install an alternative library if you wish.

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->icon('heroicon-o-document-text') // [tl! focus:start]
    ->iconColor('success') // [tl! focus:end]
    ->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .icon('heroicon-o-document-text') // [tl! focus:start]
    .iconColor('success') // [tl! focus:end]
    .send()
```



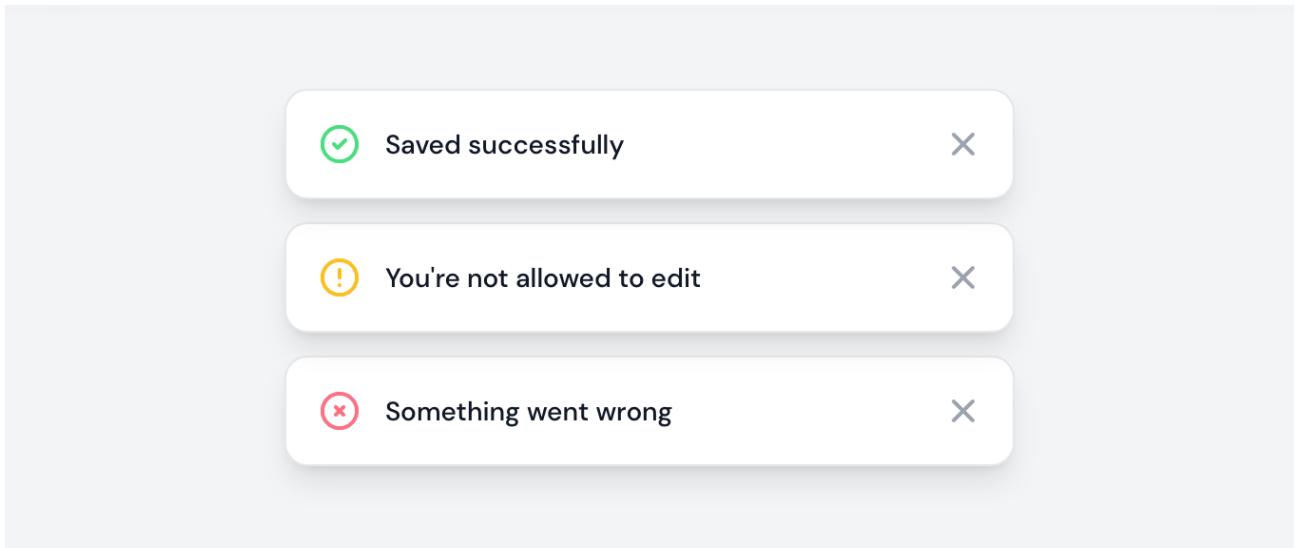
Notifications often have a status like `success`, `warning` or `danger`. Instead of manually setting the corresponding icons and colors, there's a `status()` method which you can pass the status. You may also use the dedicated `success()`, `warning()` and `danger()` methods instead. So, cleaning up the above example would look like this:

```
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success() // [tl! focus]
    ->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .success() // [tl! focus]
    .send()
```



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) // [tl! focus]
    ->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .success()
    .duration(5000) // [tl! focus]
    .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) // [tl! focus]
    ->send();
```

Or with JavaScript:

```
new Notification()
  .title('Saved successfully')
  .success()
  .seconds(5) // [tl! focus]
  .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() // [tl! focus]
  ->send();
```

Or with JavaScript:

```
new Notification()
  .title('Saved successfully')
  .success()
  .persistent() // [tl! focus]
  .send()
```

Body

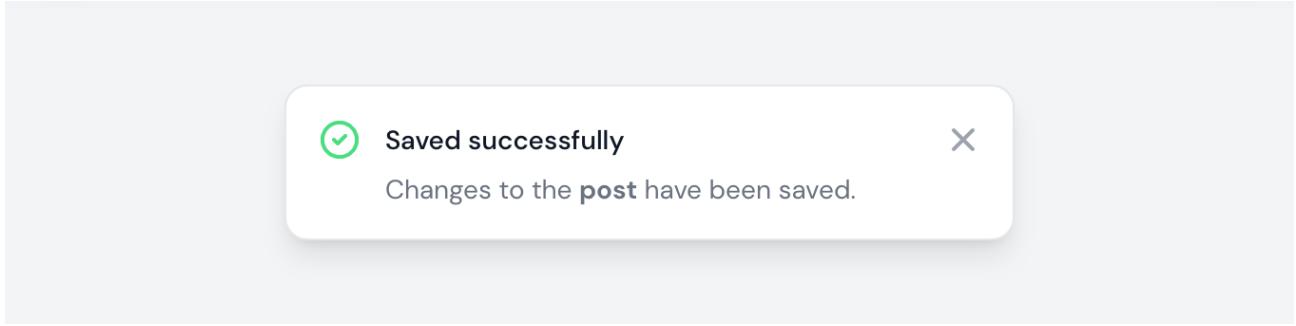
Additional notification text can be shown in the body. Similar to the title, it supports Markdown:

```
use Filament\Notifications\Notification;

Notification::make()
  ->title('Saved successfully')
  ->success()
  ->body('Changes to the **post** have been saved.') // [tl! focus]
  ->send();
```

Or with JavaScript:

```
new Notification()
  .title('Saved successfully')
  .success()
  .body('Changes to the **post** have been saved.') // [tl! focus]
  .send()
```



Actions

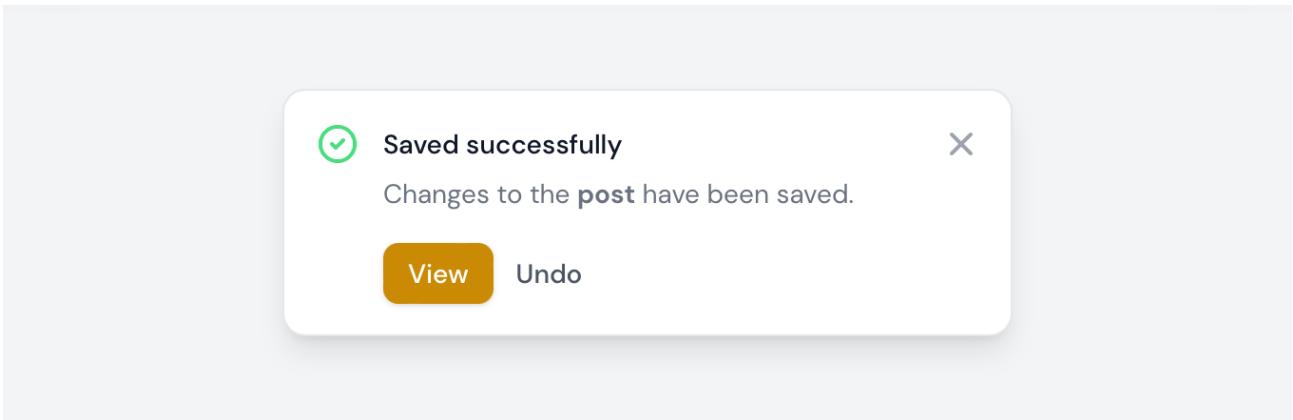
Notifications support actions that render a button or link which may open a URL or emit a Livewire event. Actions will render as link by default, but you may configure it to render a button using the `button()` method. Actions can be defined as follows:

```
use Filament\Notifications\Actions\Action; // [tl! focus]
use Filament\Notifications\Notification;

Notification::make()
    ->title('Saved successfully')
    ->success()
    ->body('Changes to the **post** have been saved.')
    ->actions([
        // [tl! focus:start]
        Action::make('view')
            ->button(),
        Action::make('undo')
            ->color('secondary'),
    ]) // [tl! focus:end]
->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .success()
    .body('Changes to the **post** have been saved.')
    .actions([
        // [tl! focus:start]
        new NotificationAction('view')
            .button(),
        new NotificationAction('undo')
            .color('secondary'),
    ]) // [tl! focus:end]
    .send()
```



Opening URLs

If clicking on an action should open a URL, optionally in a new tab, you can do so:

```
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) // [tl! focus]
        Action::make('undo')
            ->color('secondary'),
    ])
->send();
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .success()
    .body('Changes to the **post** have been saved.')
    .actions([
        new NotificationAction('view')
            .button()
            .url('/view') // [tl! focus:start]
            .openUrlInNewTab(), // [tl! focus:end]
        new NotificationAction('undo')
            .color('secondary'),
    ])
    .send()
```

Emitting events

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 emitted 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', $post), shouldOpenInNewTab: true),
        Action::make('undo')
            ->color('secondary')
            ->emit('undoEditingPost', [$post->id]), // [tl! focus]
    ])
    ->send();
```

You can also `emitSelf`, `emitUp` and `emitTo`:

```
Action::make('undo')
    ->color('secondary')
    ->emitSelf('undoEditingPost', [$post->id]) // [tl! focus]

Action::make('undo')
    ->color('secondary')
    ->emitUp('undoEditingPost', [$post->id]) // [tl! focus]

Action::make('undo')
    ->color('secondary')
    ->emitTo('another_component', 'undoEditingPost', [$post->id]) // [tl! focus]
```

Or with JavaScript:

```
new Notification()
    .title('Saved successfully')
    .success()
    .body('Changes to the **post** have been saved.')
    .actions([
        new NotificationAction('view')
            .button()
            .url('/view')
            .openUrlInNewTab(),
        new NotificationAction('undo')
            .color('secondary')
            .emit('undoEditingPost'), // [tl! focus]
    ])
    .send()
```

Similarly, `emitSelf`, `emitUp` and `emitTo` are also available:

```

new NotificationAction('undo')
    .color('secondary')
    .emitSelf('undoEditingPost') // [tl! focus]

new NotificationAction('undo')
    .color('secondary')
    .emitUp('undoEditingPost') // [tl! focus]

new NotificationAction('undo')
    .color('secondary')
    .emitTo('another_component', 'undoEditingPost') // [tl! focus]

```

Closing notifications

After opening a URL or emitting 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', $post), shouldOpenInNewTab: true),
        Action::make('undo')
            ->color('secondary')
            ->emit('undoEditingPost', [$post->id])
            ->close(), // [tl! focus]
    ])
    ->send();

```

Or with JavaScript:

```

new Notification()
    .title('Saved successfully')
    .success()
    .body('Changes to the **post** have been saved.')
    .actions([
        new NotificationAction('view')
            .button()
            .url('/view')
            .openUrlInNewTab(),
        new NotificationAction('undo')
            .color('secondary')
            .emit('undoEditingPost')
            .close(), // [tl! focus]
    ])
    .send()

```

Using the JavaScript objects

The JavaScript objects (`Notification` and `NotificationAction`) are assigned to `window.Notification` and `window.NotificationAction`, 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/module.esm'

// ...
```

Database Notifications

To start, make sure the package is [installed](#) - `@livewire('notifications')` should be in your Blade layout somewhere.

Before we start, make sure that the [Laravel notifications table](#) is added to your database:

```
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')`.

First, you must [publish the configuration file](#) for the package.

Inside the configuration file, there is a `[database]` key. To enable database notifications:

```
'database' => [
    'enabled' => true,
    // ...
],
```

Database notifications will be rendered within a modal. To open this modal, you must have a "trigger" button in your view. Create a new trigger button component in your app, for instance at [/resources/views/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 the number of unread notifications the user has.

In the configuration file, point to this new trigger view:

```
'database' => [
    'enabled' => true,
    'trigger' => 'notifications.database-notifications-trigger',
    // ...
],
```

Now, simply move the `@livewire('notifications')` component to the position in your HTML that you wish to render the database notifications trigger button. It should appear, and open the database notifications modal when clicked!

Sending 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(),
);
```

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 notifications

Without any configuration, new database notifications will only be received when the page is first loaded.

Polling

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, the configuration file polls for new notifications every 30 seconds:

```
'database' => [
    'enabled' => true,
    'polling_interval' => '30s',
    // ...
],
```

You may completely disable polling if you wish:

```
'database' => [
    'enabled' => true,
    'polling_interval' => null,
    // ...
],
```

Echo

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 emit 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));
```

Opening the 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

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 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

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's multiple options.

Styling notifications

Notifications have dedicated CSS classes you can hook into to apply your own styling:

- `filament-notifications`
- `filament-notifications-notification`
- `filament-notifications-icon`
- `filament-notifications-title`
- `filament-notifications-close-button`
- `filament-notifications-date`
- `filament-notifications-body`
- `filament-notifications-actions`

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('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-notifications::notification :notification="$notification">
    {{-- Notification content --}}
</x-notifications::notification>
```

Getters for all notification properties will be available in the view. So, a custom notification view might look like this:

```

<x-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-notifications::notification>

```

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 $data): static
    {
        return parent::fromArray()->size($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 `boot()` 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

All examples in this guide will be written using [Pest](#). However, you can easily adapt this to PHPUnit.

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.'),
        );
});
```

Chapter 5

Spatie Laravel Translatable Plugin

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

This plugin is compatible with other Filament v2.x packages.

Installation

Install the plugin with Composer:

```
composer require filament/spatie-laravel-translatable-plugin:^2.0
```

You're now ready to start [translating resources](#)!

Publishing configuration

If you wish, you may publish the configuration of the package using:

```
php artisan vendor:publish --tag=filament-spatie-laravel-translatable-plugin-config
```

Publishing translations

If you wish to translate the package, you may publish the language files using:

```
php artisan vendor:publish --tag=filament-spatie-laravel-translatable-plugin-translations
```

Upgrading

To upgrade the package to the latest version, you must run:

```
composer update
```

Getting Started

This guide assumes that you've already set up your model to be translatable as per [Spatie's documentation](#).

Preparing your resource class

First, you must apply the `Filament\Resources\Concerns\Translatable` trait to your resource class. This adds a `getTranslatableLocales()` method, which you may override to return an array of locales that the resource can be translated into:

```
use Filament\Resources\Concerns\Translatable;
use Filament\Resources\Resource;

class BlogPostResource extends Resource
{
    use Translatable;

    // ...

    public static function getTranslatableLocales(): array
    {
        return ['en', 'es'];
    }
}
```

You may [publish the package's configuration file](#) to set the `default_locales` for all resources at once.

Making resource pages translatable

After [preparing your resource class](#), you must make each of your resource's pages translatable too. You can find your resource's pages in the `Pages` directory of each resource folder:

```
use Filament\Pages\Actions;
use Filament\Resources\Pages\ListRecords;

class ListBlogPosts extends ListRecords
{
    use ListRecords\Concerns\Translatable;

    protected function getActions(): array
    {
        return [
            Actions\LocaleSwitcher::make(),
            // ...
        ];
    }

    // ...
}
```

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

class CreateBlogPost extends CreateRecord
{
    use CreateRecord\Concerns\Translatable;

    protected function getActions(): array
    {
        return [
            Actions\LocaleSwitcher::make(),
            // ...
        ];
    }

    // ...
}
```

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

class EditBlogPost extends EditRecord
{
    use EditRecord\Concerns\Translatable;

    protected function getActions(): array
    {
        return [
            Actions\LocaleSwitcher::make(),
            // ...
        ];
    }

    // ...
}
```

And if you have a `ViewRecord` page for your resource:

```
use Filament\Pages\Actions;
use Filament\Resources\Pages\ViewRecord;

class ViewBlogPost extends ViewRecord
{
    use ViewRecord\Concerns\Translatable;

    protected function getActions(): array
    {
        return [
            Actions\LocaleSwitcher::make(),
            // ...
        ];
    }

    // ...
}
```

Chapter 6

Spatie Laravel Tags Plugin

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

This plugin is compatible with other Filament v2.x packages.

Installation

Install the plugin with Composer:

```
composer require filament/spatie-laravel-tags-plugin:"^2.0"
```

If you haven't already done so, you need to publish the migration to create the tags table:

```
php artisan vendor:publish --provider="Spatie\Tags\TagsServiceProvider" --tag="tags-migrations"
```

Run the migrations:

```
php artisan migrate
```

You must also prepare your Eloquent model for attaching tags.

For more information, check out [Spatie's documentation](#).

You're now ready to start using the form components and table columns!

Upgrading

To upgrade the package to the latest version, you must run:

```
composer update
```

Form Components

This guide assumes that you've already set up your model attach tags as per [Spatie's documentation](#).

You may use the field in the same way as the [original tags input](#) field:

```
use Filament\Forms\Components\SpatieTagsInput;

SpatieTagsInput::make('tags'),
```

*The field will automatically load and save its tags to your model. To set this functionality up, **you must also follow the instructions set out in the [field relationships section](#)**. If you're using the [admin panel](#), you can skip this step.*

Optionally, you may pass a `type()` allows you to group tags into collections:

```
use Filament\Forms\Components\SpatieTagsInput;

SpatieTagsInput::make('tags')->type('categories'),
```

The tags input supports all the customization options of the [original tags input component](#).

Table Columns

This guide assumes that you've already set up your model attach tags as per [Spatie's documentation](#).

To use the tags column:

```
use Filament\Tables\Columns\SpatieTagsColumn;

SpatieTagsColumn::make('tags'),
```

Optionally, you may pass a `type()`:

```
use Filament\Tables\Columns\SpatieTagsColumn;

SpatieTagsColumn::make('tags')->type('categories'),
```

The `type` allows you to group tags into collections.

The tags column supports all the customization options of the [original tags column](#).

Chapter 7

Spatie Laravel Settings Plugin

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

This plugin is compatible with other Filament v2.x packages.

Installation

Install the plugin with Composer:

```
composer require filament/spatie-laravel-settings-plugin:"^2.0"
```

You're now ready to start building [settings pages!](#)

Upgrading

To upgrade the package to the latest version, you must run:

```
composer update
```

Getting Started

Preparing your page class

Settings pages are Filament pages that extend the `Filament\Pages\SettingsPage` class.

This package uses the `spatie/laravel-settings` package to store and retrieve settings via the database.

Before you start, create a settings class in your `app/Settings` directory, and a database migration for it. You can find out more about how to do this in the [Spatie documentation](#).

Once you've created your settings class, you can create a settings page in Filament for it using the following command:

```
php artisan make:filament-settings-page ManageFooter FooterSettings
```

In this example, you have a `FooterSettings` class in your `app/Settings` directory.

In your new settings page class, generated in the `app/Filament/Pages` directory, you will see the static `$settings` property assigned to the settings class:

```
protected static string $settings = FooterSettings::class;
```

Building a form

You must define a form schema to interact with your settings class inside the `getFormSchema()` method.

Since the `form builder` is installed in the admin panel by default, you may use any form `fields` or `layout components` you like:

```
use Filament\Forms\Components\Repeater;
use Filament\Forms\Components\TextInput;

protected function getFormSchema(): array
{
    return [
        TextInput::make('copyright')
            ->label('Copyright notice')
            ->required(),
        Repeater::make('links')
            ->schema([
                TextInput::make('label')->required(),
                TextInput::make('url')
                    ->url()
                    ->required(),
            ]),
    ];
}
```

The name of each form field must correspond with the name of the property on your settings class.

The form will automatically be filled with settings from the database, and saved without any extra work.

Chapter 8

Spatie Laravel Media Library Plugin

Installation

Requirements

Filament has a few requirements to run:

- PHP 8.0+
- Laravel v8.0+
- Livewire v2.0+

This plugin is compatible with other Filament v2.x packages.

Installation

Install the plugin with Composer:

```
composer require filament/spatie-laravel-media-library-plugin:^2.0
```

If you haven't already done so, you need to publish the migration to create the media table:

```
php artisan vendor:publish --provider="Spatie\MediaLibrary\MediaLibraryServiceProvider" --tag="migrations"
```

Run the migrations:

```
php artisan migrate
```

You must also prepare your Eloquent model for attaching media.

For more information, check out [Spatie's documentation](#).

You're now ready to start using the form components and table columns!

Upgrading

To upgrade the package to the latest version, you must run:

```
composer update
```

Form Components

You may use the field in the same way as the [original file upload](#) field:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('avatar'),
```

The media library file upload supports all the customization options of the [original file upload component](#).

*The field will automatically load and save its uploads to your model. To set this functionality up, **you must also follow the instructions set out in the [field relationships](#) section.** If you're using the [admin panel](#), you can skip this step.*

Passing a collection

Optionally, you may pass a `collection()` allows you to group files into categories:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('avatar')->collection('avatars'),
```

Reordering files

In addition to the behaviour of the normal file upload, Spatie's Media Library also allows users to reorder files.

To enable this behaviour, use the `enableReordering()` method:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')
    ->multiple()
    ->enableReordering(),
```

You may now drag and drop files into order.

Adding custom properties

You may pass in [custom properties](#) when uploading files using the `customProperties()` method:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')
    ->multiple()
    ->customProperties(['zip_filename_prefix' => 'folder/subfolder/']),
```

Generating responsive images

You may [generate responsive images](#) when the files are uploaded using the `responsiveImages()` method:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')
    ->multiple()
    ->responsiveImages(),
```

Using conversions

You may also specify a `conversion()` to load the file from showing it in the form, if present:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')->conversion('thumb'),
```

Storing conversions on a separate disk

You can store your conversions and responsive images on a disk other than the one where you save the original file. Pass the name of the disk where you want conversion to be saved to the `conversionsDisk()` method:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')->conversionsDisk('s3'),
```

Storing media-specific manipulations

You may pass in manipulations that are run when files are uploaded using the `manipulations()` method:

```
use Filament\Forms\Components\SpatieMediaLibraryFileUpload;

SpatieMediaLibraryFileUpload::make('attachments')
    ->multiple()
    ->manipulations([
        'thumb' => ['orientation' => '90'],
    ]),
```

Table Columns

To use the media library image column:

```
use Filament\Tables\Columns\SpatieMediaLibraryImageColumn;

SpatieMediaLibraryImageColumn::make('avatar'),
```

The media library image column supports all the customization options of the [original image column](#).

Passing a collection

Optionally, you may pass a `collection()`:

```
use Filament\Tables\Columns\SpatieMediaLibraryImageColumn;

SpatieMediaLibraryImageColumn::make('avatar')->collection('avatars'),
```

The `collection` you to group files into categories.

Using conversions

You may also specify a `conversion()` to load the file from showing it in the form, if present:

```
use Filament\Tables\Columns\SpatieMediaLibraryImageColumn;

SpatieMediaLibraryImageColumn::make('avatar')->conversion('thumb'),
```