Features

Descrição e usabilidade das features do painel Labs.

Ordenação

Introdução

Esta feature foi desenvolvida para fornecer um **sistema de ordenação flexível e reutilizável** para os módulos do painel.
O objetivo é simplificar a implementação de ordenações, padronizando a forma como os registros são organizados e garantindo consistência entre diferentes módulos, sem a necessidade de reescrever a lógica de movimentação ou atualização de posições.

O sistema suporta **ordenções simples**, que envolvem apenas registros do mesmo módulo, permitindo que o usuário reorganize os itens de forma direta e intuitiva.
Além disso, ele também é capaz de lidar com **cenários mais complexos**, envolvendo múltiplos níveis de relacionamento entre modelos diferentes. Por exemplo, em um módulo que gerencia seções e produtos, é possível ordenar os produtos **dentro de uma seção específica** (*seção → produto*). Nesse caso, o sistema primeiro aplica filtros para selecionar apenas os registros do nível superior (a seção) e, em seguida, realiza a ordenação dos registros do nível inferior (os produtos).

Outro ponto importante é o suporte a **ordenação recursiva dentro da própria model**, útil em casos onde um registro pode ter outro registro do mesmo tipo como “pai”. Esse recurso permite criar estruturas hierárquicas complexas, como categorias e subcategorias, menus com itens filhos ou qualquer outro cenário em que a ordenação precise percorrer múltiplos níveis da mesma tabela, gerando uma **estrutura de ordenação potencialmente infinita**.

Com essa abordagem, qualquer módulo do painel pode adotar o sistema de ordenação de forma rápida e confiável, aproveitando toda a lógica de backend e frontend já pronta, garantindo consistência, performance e facilidade de manutenção.

Model e Registros

Para que o sistema de ordenação funcione corretamente, cada **model** envolvida deve seguir uma estrutura mínima obrigatória.
Abaixo estão descritas as configurações e implementações necessárias para garantir a compatibilidade com a feature.

---

1. **Trait obrigatória**

Toda model que utilizar o sistema de ordenação **deve incluir a trait**:

use App\Traits\SortableTrait;
Essa trait contém os métodos e comportamentos essenciais para manipular a posição dos registros, incluindo funções de ordenação, reordenação após exclusões e cálculo da próxima posição disponível.

---

2. **Evento deleted com reorder()**

É obrigatório que a model implemente o evento **deleted**, chamando o método **reorder()** após a exclusão de um registro.
Isso garante que a ordenação seja recalculada automaticamente, mantendo a sequência correta dos itens.

Exemplo:

```
protected static function boot()
    {
        parent::boot();

        static::deleted(function ($model) {
            $model->reorder(); // executa após deletar com sucesso
        });
    }
```

> **Observação:**
> Se a ordenação possuir níveis (ex: seção → produto), é necessário passar um *filtro* via **Closure** como terceiro parâmetro na função **reorder()**, indicando o escopo onde a reordenação deve ocorrer (por exemplo, dentro da mesma seção).

---

3. **Relacionamentos entre níveis**

Quando o módulo utiliza **ordenação com múltiplos níveis**, cada nível adicional deve possuir uma **relação "belongsTo"** com o nível imediatamente anterior.
Essa relação é obrigatória para que o sistema saiba a qual contexto o registro pertence e consiga filtrar corretamente antes de aplicar a ordenação.

Exemplo:
Na model do **nível 3**, deve existir uma relação com o **nível 2**:

```
public function nivel2() { return $this->belongsTo(NivelDoi::class, 'nivel_2_id'); }
```

---

4. **Colunas obrigatórias na tabela**

A tabela da model deve conter as seguintes colunas:

Coluna	Tipo	Obrigatoriedade	Descrição
order	**integer**	**Obrigatória**	Define a posição do registro dentro do contexto da ordenação.
parent_id	**integer**	Obrigatória **somente** quando o tipo de ordenação é **recursiva** (entre registros da mesma model).	
<nome_da_fk>	**integer**	Obrigatória **quando existe mais de um nível** (não recursiva). Representa a foreign key que liga ao nível anterior.	

> **Exemplo:**
>
> - Em uma ordenação recursiva (ex: categorias com subcategorias), use **parent_id**.

> - Em uma ordenação com múltiplos níveis (ex: seção → produto), use a FK correspondente, como **secao_id**.
> - Em uma ordenação simples (sem níveis), nenhuma FK adicional é necessária.

Configuração no controller

Para habilitar o sistema de ordenação em um módulo, é necessário implementar no **controller** uma função pública chamada **orderConfig()**
Essa função é responsável por retornar duas arrays principais que controlam todo o comportamento da ordenação: **$generalConfig e $configLevels**.

$generalConfig

A array **$generalConfig** contém configurações gerais da ordenação.
Atualmente, possui as seguintes propriedades possíveis:

```
$generalConfig = ['selfGroupedRecords' => false];
```

Propriedade	Tipo	Descrição
view	**string | null**	Para alterar a view que será exibida a ordenação, pode ser passado um nome de rota (por exemplo, "*panel.layouts.modules.order*")
selfGroupedRecords	**bool**	Define se a ordenação será **recursiva dentro do mesmo módulo** (por exemplo, quando um registro pode ter outro como “pai”).

$configLevels

A array **$configLevels** é onde é definida toda a **lógica de ordenação**.
Cada item dentro dela representa um **nível de ordenação** — ou seja:

- Se for uma ordenação **simples** (sem níveis adicionais ou recursivos), haverá **apenas um nível**.
- Se for uma ordenação **hierárquica** (por exemplo, *seção → produto*), haverá **um nível para cada relação**.

> **Observações:**
>
> - A **ordem dos itens na array** define a hierarquia dos níveis.
> O **primeiro item** representa o **primeiro nível** da ordenação, o **segundo item** o **segundo nível**, e assim por diante.

> - A chave de cada nível deve ser um **identificador em camelCase**, de preferência o nome do módulo.

Cada nível da configuração deve conter as seguintes propriedades:

```
$configLevels = [

        'levelOne' => [
            'label' => 'Nivel 1',
            'model' => SubmodulesTest::class,
            'sortable' => true,
            'labelBtnNextLevel' => 'Ordernar itens do nível 1',
            'breadcrumbColumnName' => 'name',
            'order' => ['column' => 'order', 'type' => 'ASC'],
            'AdditionalQuery' => function ($q) { $q->where('id', 1); },
            'columnsToList' => [
                [
                    'name' => 'active',
                    'label' => 'Status'
                ],
                [
                    'name' => 'id',
                    'label' => '#'
                ],
                [
                    'name' => 'name',
                    'label' => 'Título'
                ],
            ],
        ],
        'levelTwo' => [
            'label' => 'Nivel 2',
            'model' => NivelDoi::class,
            'sortable' => true,
            'order' => ['column' => 'order', 'type' => 'ASC'],
            'breadcrumbColumnName' => 'name',
            'columnsToList' => [
                [
                    'name' => 'active',
```

```
                'label' => 'Status'
            ],
            [
                'name' => 'id',
                'label' => '#'
            ],
            [
                'name' => 'name',
                'label' => 'Título'
            ],
        ],
    ],
    // Adicione mais níveis aqui, se necessário
];
```

Propriedade	Tipo	Descrição
label	**string**	Nome descritivo do nível.
model	**Model**	Instância da model correspondente ao nível.
sortable	**bool**	Define se o nível atual pode ser ordenado (**true** / **false**).
labelBtnNextLevel	**string** *(opcional)*	Texto exibido no botão que permite avançar para o próximo nível.
breadcrumbColumnName	**string**	Coluna usada para exibir o nome no breadcrumb.
order	**array**	Define a ordenação padrão da listagem (**['column' => 'id', 'type' => 'asc']**).
columnToList	**array**	Define as colunas exibidas na listagem de ordenação. Cada item deve conter **name** (coluna no banco) e **label** (rótulo exibido no frontend).
additionalQuery	**closure** *(opcional)*	Permite aplicar filtros adicionais na query de listagem.

> **Observação:**
>
> Na propriedade **columnToList**, é possível exibir campos que venham de relações da model.

> Para isso, basta referenciar o campo desejado como **string**, utilizando o nome da relação seguido do nome da coluna.
>
> **Exemplo:**
>
> ```
> ['name' => 'parent->title', 'label' => 'Seção Pai']
> ```
>
> Nesse caso, **parent** representa o nome da relação definida na model, e **nome** é o campo que será exibido no componente de ordenação.

Exemplo:

```
public function orderConfig()
    {
        $generalConfig = ['selfGroupedRecords' => false];

        $configLevels = [

            'author' => [
                'label' => 'Autores',
                'model' => Author::class,
                'sortable' => true,
                'order' => ['column' => 'order', 'type' => 'ASC'],
                'breadcrumbColumnName' => 'title',
                'columnsToList' => [
                    [
                        'name' => 'active',
                        'label' => 'Status'
                    ],
                    [
                        'name' => 'id',
                        'label' => '#'
                    ],
                    [
                        'name' => 'name',
                        'label' => 'Título'
                    ],
                ],
```

```
            ],
            // Adicione mais níveis aqui, se necessário
        ];

        return ['generalConfig' => $generalConfig, 'configLevels' => $configLevels];
    }
```

Definindo a ordem no store

No método **store** (ou equivalente) do controller, a coluna **order** deve ser preenchida automaticamente com o **próximo valor de ordem** disponível, utilizando o método getNextOrder() da trait.

Esse método retorna o próximo valor de posição com base no contexto atual.
Se for necessário aplicar um filtro (por exemplo, ordenar apenas dentro de um determinado nível), o método aceita uma **Closure** como segundo parâmetro:

Exemplo:

Nesse caso, o sistema calcula automaticamente o próximo valor com base em todos os registros existentes no módulo atual.

```
$model->order = $model->getNextOrder();
```

Exemplo com filtro de nível (recomendado para estruturas hierárquicas):

Quando a ordenação depende de outro nível — como no caso **seção → produto**, em que os produtos devem ser ordenados apenas dentro de uma determinada seção —
é obrigatório utilizar uma **Closure** para filtrar o contexto antes de calcular a próxima posição.

```
$model->order = $model->getNextOrder(function ($query) use ($secao_id) {
    return $query->where('secao_id', $secao_id);
});
```

Rotas e View

Para que o sistema de ordenação funcione corretamente em um módulo, é necessário definir duas **rotas principais** responsáveis por exibir e processar a ordenação.
Essas rotas devem ser criadas dentro do **grupo de rotas do módulo** e vinculadas ao seu respectivo **controller**.

Rotas

Os módulos que implementarem a ordenação precisam conter as seguintes rotas:

Descrição das rotas:

Método	Caminho	Controller / Método	Descrição
GET	**/ordenar**	**order()**	Responsável por carregar a view principal da ordenação e inicializar o componente com base nas configurações definidas no controller.
POST	**/ordenar**	**processOrder()**	Recebe a nova ordem enviada pelo frontend e processa a atualização dos registros no banco de dados.

Ambas as rotas utilizam o middleware "*panel.check.module.permission:update*", garantindo que apenas usuários com permissão de **edição** possam acessar e modificar a ordem dos registros.

Exemplo:

```
Route::get('/ordernar', [AuthorController::class, 'order'])
->name('order')->middleware('panel.check.module.permission:update');


Route::post('/ordernar', [AuthorController::class, 'processOrder'])
->name('processOrder')->middleware('panel.check.module.permission:update');
```

View

Por padrão, o sistema de ordenação utiliza uma **view global** compartilhada entre todos os módulos, localizada em:

"panel.layouts.modules.order"

Essa view é responsável por renderizar a interface do sistema de ordenação, exibindo os registros conforme as configurações definidas no método **orderConfig()** do controller.
Ela já contém toda a estrutura necessária para:

- Exibir os níveis de ordenação e breadcrumbs;
- Listar os registros conforme o nível atual;
- Enviar a nova ordem para o backend através da rota **processOrder**;
- Atualizar dinamicamente a interface conforme o usuário interage com os níveis.

Personalização da View

Embora exista uma view global, o sistema permite **substituí-la por uma view personalizada**, caso o módulo necessite de um layout ou comportamento específico.
Para isso, basta informar o **caminho da nova view** dentro da configuração **$generalConfig** do método **orderConfig()**.

Exemplo:

```
$generalConfig = [ 'selfGroupedRecords' => false, 'view' => 'panel.modules.produtos.orderCustom', ];
```

Nesse caso, o sistema utilizará a view "*panel.modules.produtos.orderCustom*" em vez da padrão "*panel.layouts.modules.order*".

> **Observação:**
> A view personalizada deve seguir a mesma estrutura básica da view padrão — ou seja, deve conter o componente responsável por interagir com o sistema de ordenação e enviar as alterações para o backend.
> Dessa forma, mantém-se a compatibilidade com a lógica existente, mesmo quando o layout é adaptado.

Toggle Field

Introdução

A feature **Toggle Field** permite **controlar a exibição de campos do formulário de forma dinâmica** com base no valor de outro campo (por exemplo, exibir **CPF** se o tipo de pessoa for **F**, ou **CNPJ** se for **J**).

Ela foi criada para integrar o **Laravel (backend)** e o **Blade (frontend)**, oferecendo um controle automatizado de:

- Exibição condicional de campos;
- Filtragem automática das validações conforme os campos visíveis.

A configuração é feita através de **Enums** que implementam a interface **FeatureToggleFieldInterface**.

Como Criar e Configurar o Enum

Estrutura da Interface

```
namespace App\Interfaces;


interface FeatureToggleFieldInterface
{
    public function visibleFields();
    public static function getDynamicFields();
    public static function buildJsonStructure(): array;
}
```

Essa interface define o contrato que os Enums precisam seguir para informar quais campos são visíveis conforme o valor selecionado.

Crie um Enum que implemente **FeatureToggleFieldInterface**.
Cada caso do Enum representa uma variação do campo controlador (por exemplo, tipo de pessoa, tipo de página, etc).

Exemplo:

```
namespace App\Enums\Panel;


use App\Interfaces\FeatureToggleFieldInterface;


enum ContentPageTypeEnum: int implements FeatureToggleFieldInterface
{
    case PADRAO = 1;
```

```
case SERVICOS = 2;
case PRODUTOS = 3;

public function label(): string
{
    return match ($this) {
        static::PADRAO => 'Padrão',
        static::SERVICOS => 'Serviços',
        static::PRODUTOS => 'Produtos',
    };
}

public function visibleFields(): array
{
    return match ($this) {
        static::PADRAO => ['parent_page_id'],
        static::SERVICOS => [],
        static::PRODUTOS => ['parent_page_id'],
    };
}

public static function getDynamicFields(): array
{
    $allFields = [];
    foreach (self::cases() as $case)
        $allFields = array_merge($allFields, $case->visibleFields());

    return array_values(array_unique($allFields));
}

public static function buildJsonStructure(): array
{
    $instances = [];

    foreach (self::cases() as $case) {
        $instances[$case->value] = [
            'visibleFields' => $case->visibleFields()
        ];
    }
```

```
        return ['instances' => $instances];
    }
}
```

□□ Importante

- O método **visibleFields()** retorna os **names** dos campos que devem aparecer para cada caso.
- Se um campo aparece em pelo menos um caso, ele é considerado parte da feature Toggle Field.
- Campos que não aparecem em nenhum caso são ignorados pelo comportamento da feature.

Configuração no Controller

No método **create** ou **edit**, gere a estrutura JSON e envie para a view:

```
public function create()
{
    $togglePageTypeStructure = ContentPageTypeEnum::buildJsonStructure();

    return view(module()->view('form'), [
      'togglePageTypeStructure' => $togglePageTypeStructure,
    ]);
}
```

Na hora de validar o request (**store** ou **update**), passe a instância do Enum no método de validação:

```
public function update(Request $request, ContentPage $contentPage)
{
    $this->validateRequest($request, ContentPageTypeEnum::tryFrom($request->type));
}
```

Também é possível passar um **array de Enums**, caso o comportamento envolva mais de um campo controlador:

```
$this->validateRequest($request, [
    ContentPageTypeEnum::tryFrom($request->type),
    AnotherToggleEnum::tryFrom($request->other_type)
]);
```

Definição na View

Na view **Blade**, siga estas regras:

1. **Adicione a classe *feature-toggle-field*** nos campos (ou containers) que devem ser controlados.
2. Para cada Enum que controla uma parte da tela, adicione uma linha JS chamando o método **addToggleField.**

Exemplo:

```
<select name="type" id="type">
    <option value="1">Padrão</option>
    <option value="2">Serviços</option>
    <option value="3">Produtos</option>
</select>

<div class="feature-toggle-field" data-name="parent_page_id">
    <label for="parent_page_id">Página Pai</label>
    <input type="text" name="parent_page_id" id="parent_page_id">
</div>

<script>
    toggleField.addToggleField('type', @json($togglePageTypeStructure));
</script>
```

Explicação

- O **primeiro parâmetro** ('type') é o nome do campo que executa o toggle (pode ser um **radio** ou **select**).
- O **segundo parâmetro** é o **JSON** retornado pelo Enum através de **buildJsonStructure()** .
- É possível adicionar várias chamadas de **addToggleField()** caso existam múltiplos toggles independentes na mesma página.