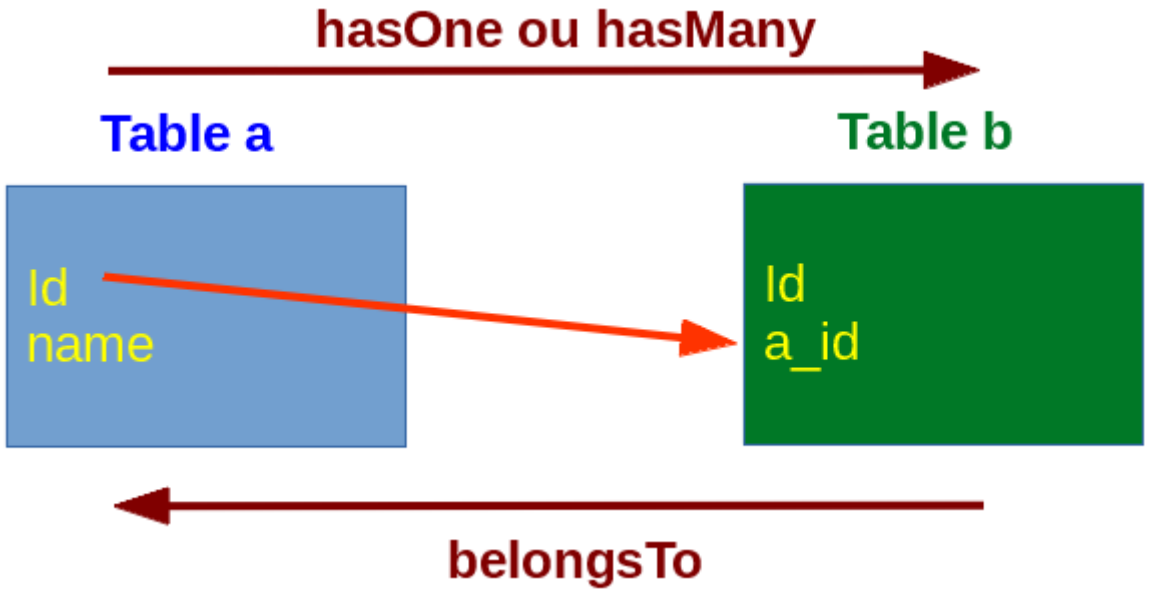
Cours Laravel 5.5 – les données – le polymorphisme

Lors des deux précédents chapitres on a vu les principales relations que nous offre Eloquent : **hasMany** et **belongsTomany**. Je ne vous ai pas parlé de la relation **hasOne** parce que c'est juste du **hasMany** limité à un seul enregistrement et est peu utilisé. Dans tous les cas qu'on a vus on considère 2 tables en relation. Dans le présent chapitre on va envisager le cas où une table peut être en relation avec plusieurs autres tables, ce qui se nomme du polymorphisme.

Un peu de théorie

La relation 1:1 ou 1:n

On a vu cette relation, en voici une schématisation pour fixer les esprits :

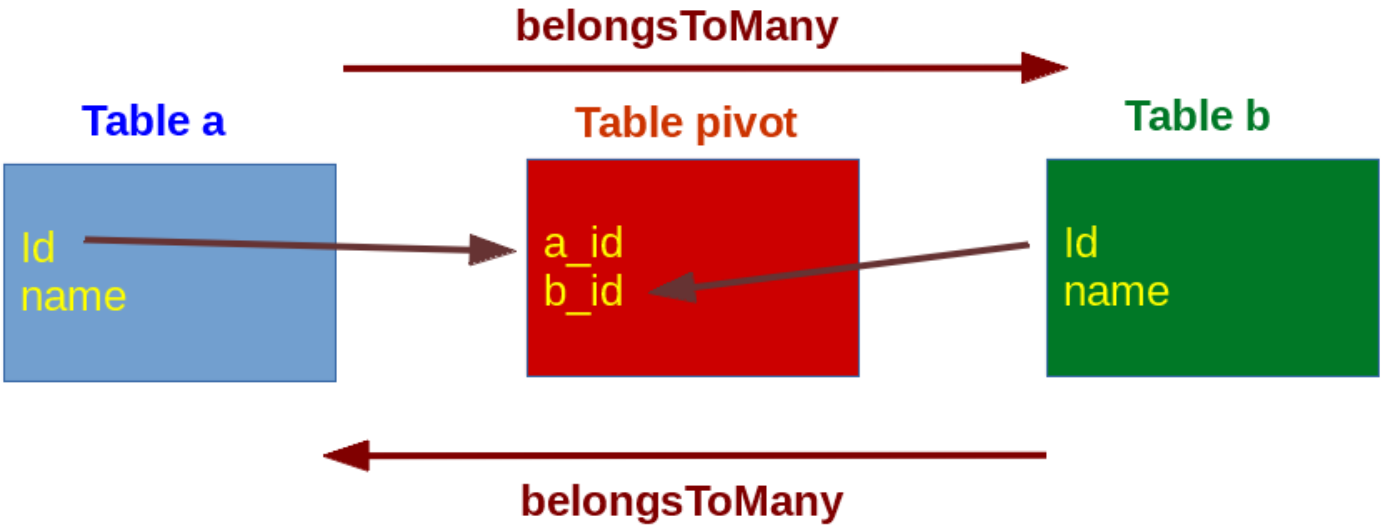


On a **a possède un b** (**hasOne**) ou **a possède plusieurs b** (**hasMany**).

La réciproque : **b est possédé par un a** (**belongsTo**).

La relation n:n

On a vu aussi cette relation, en voici une schématisation pour fixer les esprits :



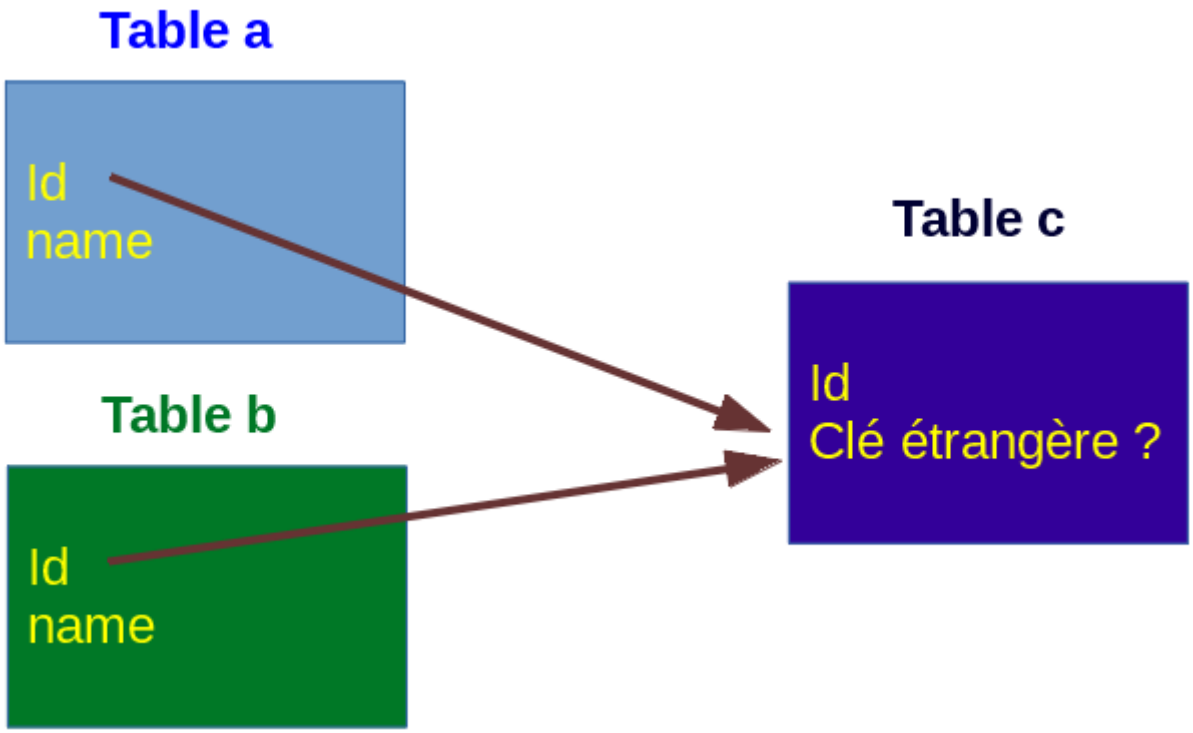
On a **a appartient à un ou plusieurs b** (**belongsToMany**).

Et on a **b appartient à un ou plusieurs a** (**belongsToMany**).

La relation une table vers plusieurs tables

Type de relation 1:n

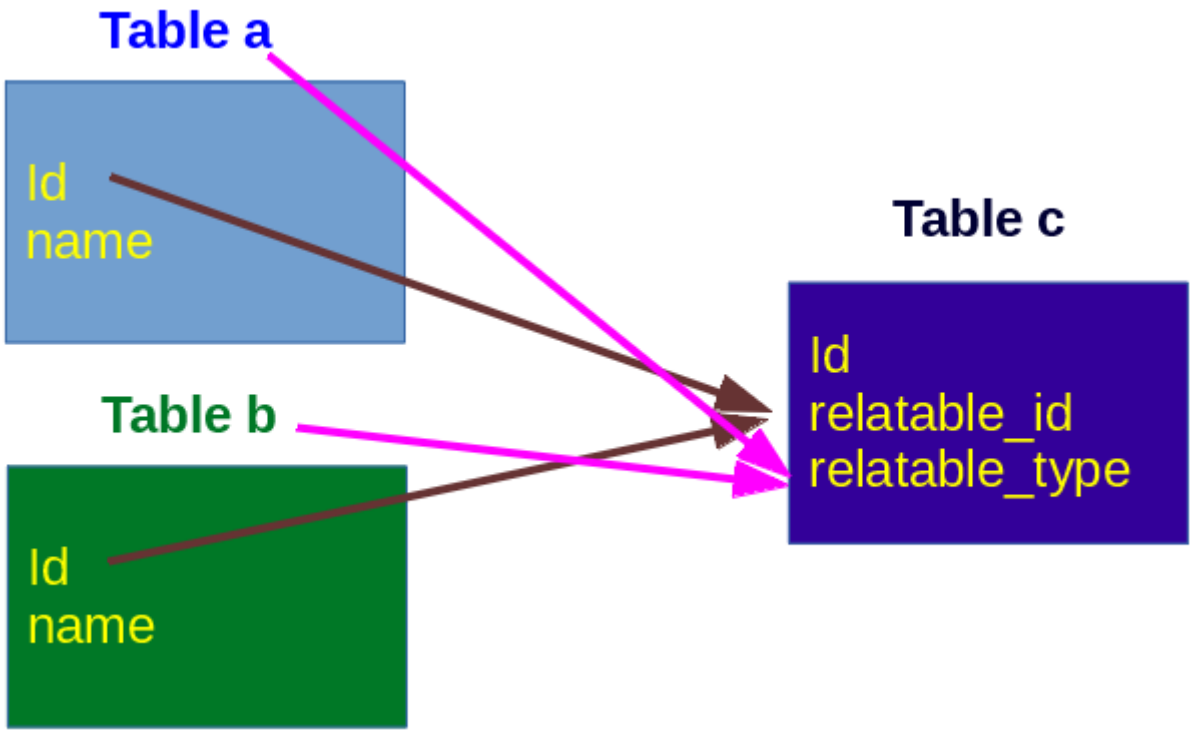
Maintenant imaginons cette situation :



La table c peut être en relation soit avec la table a, soit avec la table b. Dans cette situation comment gérer une clé étrangère dans la table c ? Comment l'appeler et comment savoir avec quelle table elle est en relation ?

On voit bien qu'il va falloir une autre information : **connaître sûrement la table en relation**.

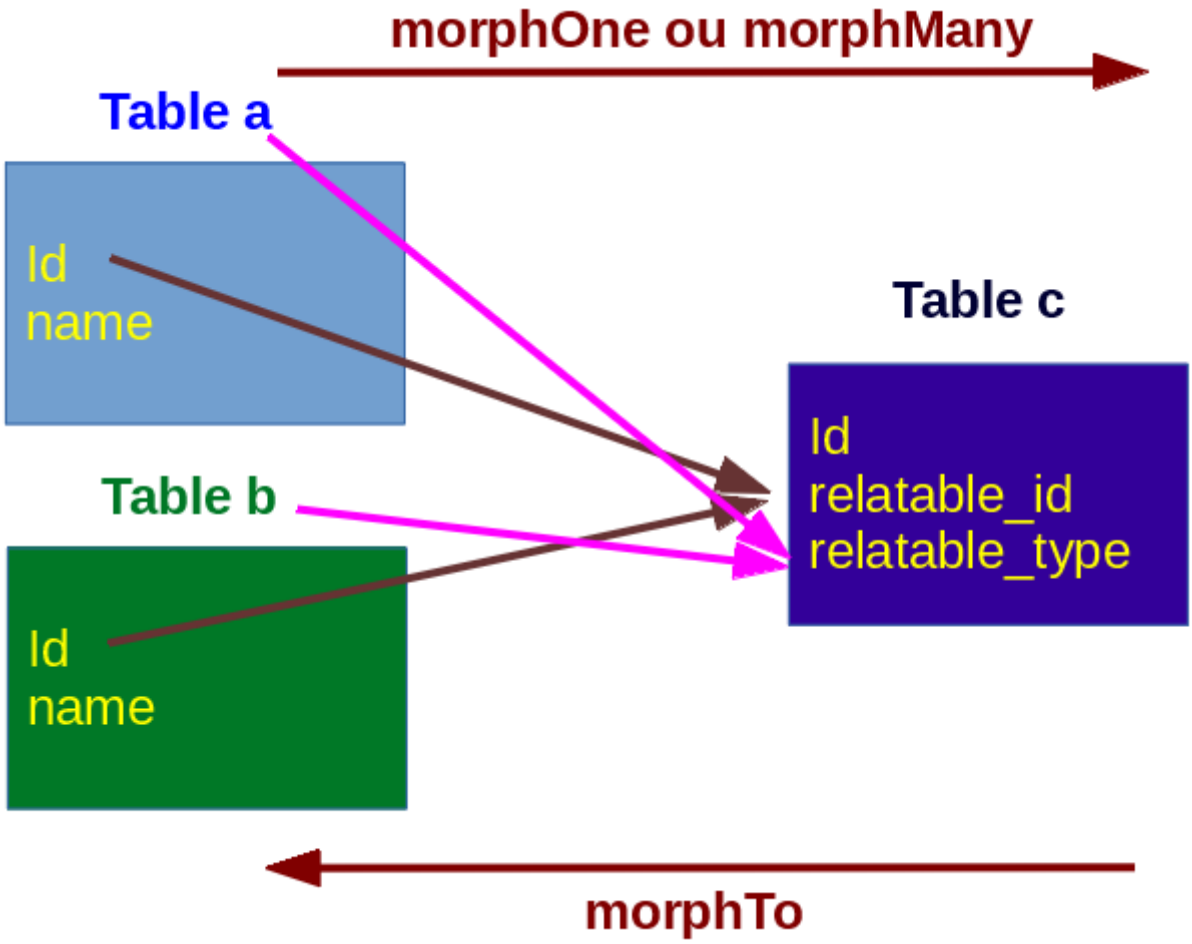
Puisqu'on a besoin de deux informations il nous faut deux colonnes :



On a donc deux colonnes :

- **relatable_id** : la clé étrangère qui mémorise l'identifiant de l'enregistrement en relation
- **relatable_type** : la classe du modèle en relation.

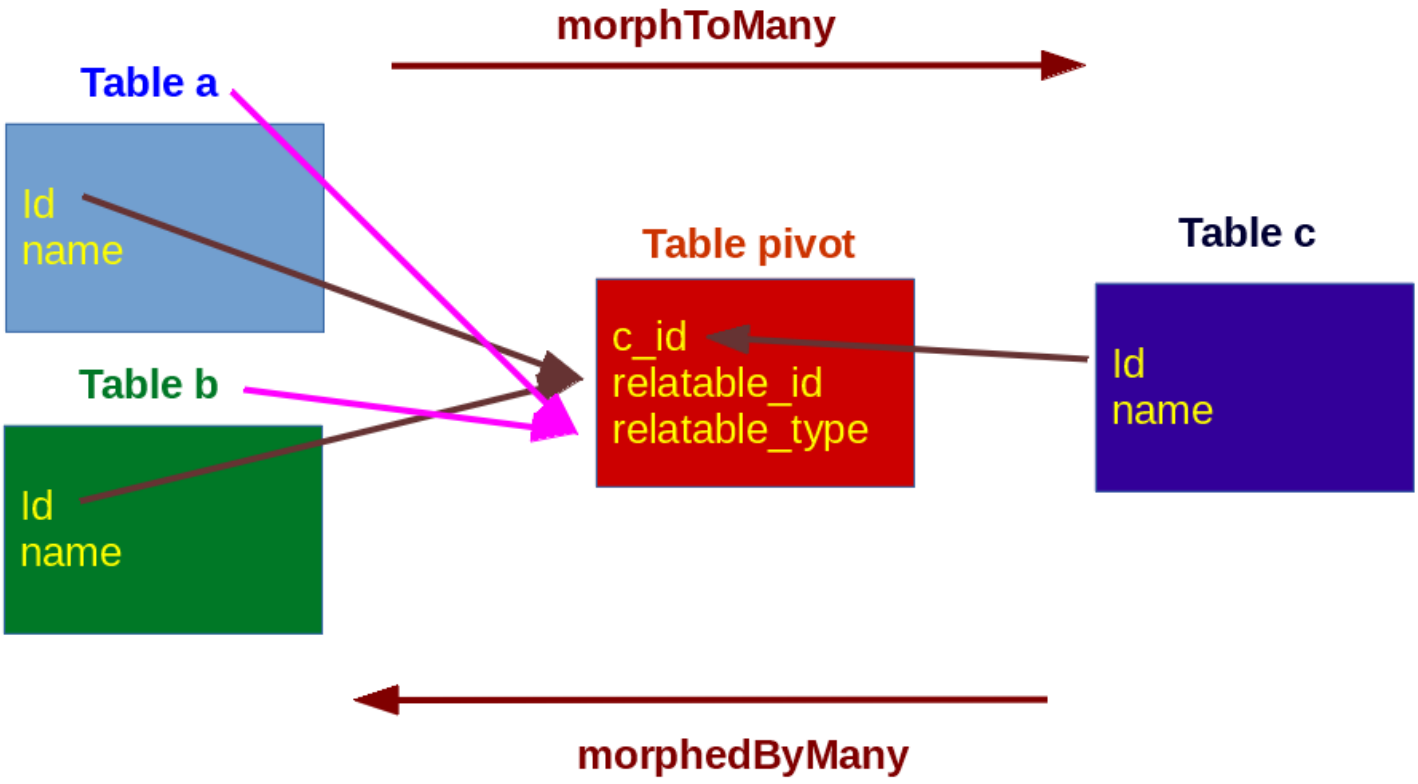
Voici la figure complétée avec les noms de ces relations :



- **morphOne** : c'est le **hasOne** mais issu de plusieurs tables.
- **morphMany** : c'est le **hasMany** mais issu de plusieurs tables.
- **morphTo** : c'est le **belongsTo** mais à destination de plusieurs tables.

Type de relation n:n

On peut avoir le même raisonnement pour une relation de type **n:n** avec plusieurs tables d'un côté de la relation :



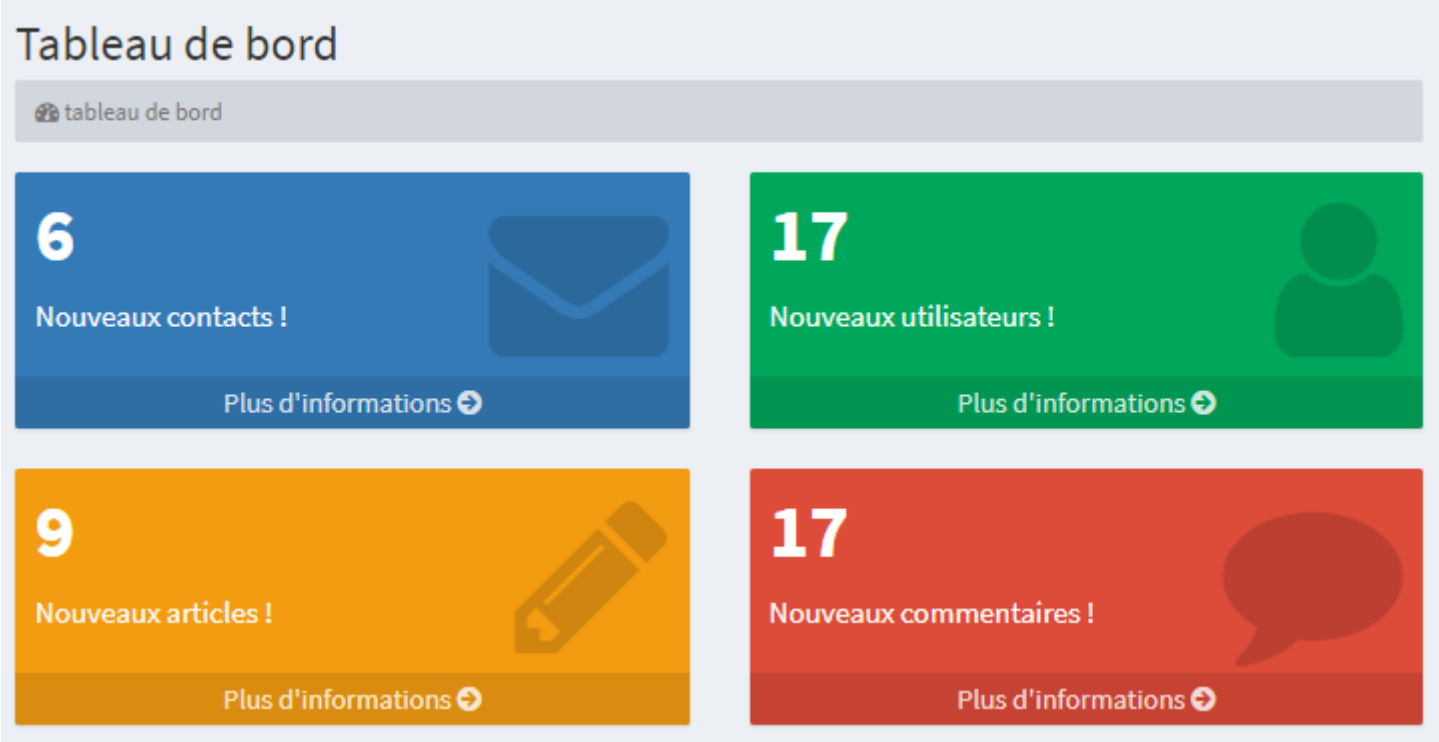
- **morphToMany** : c'est le **belongsToMany** mais issu de plusieurs tables.
- **morphedByMany** : c'est le **belongsToMany** mais en direction de plusieurs tables.

L'application d'exemple

Maintenant qu'on a vu la théorie passons à la pratique avec un cas dans l'application d'exemple.

Présentation

Quand vous allez dans l'administration vous tombez sur le tableau de bord :



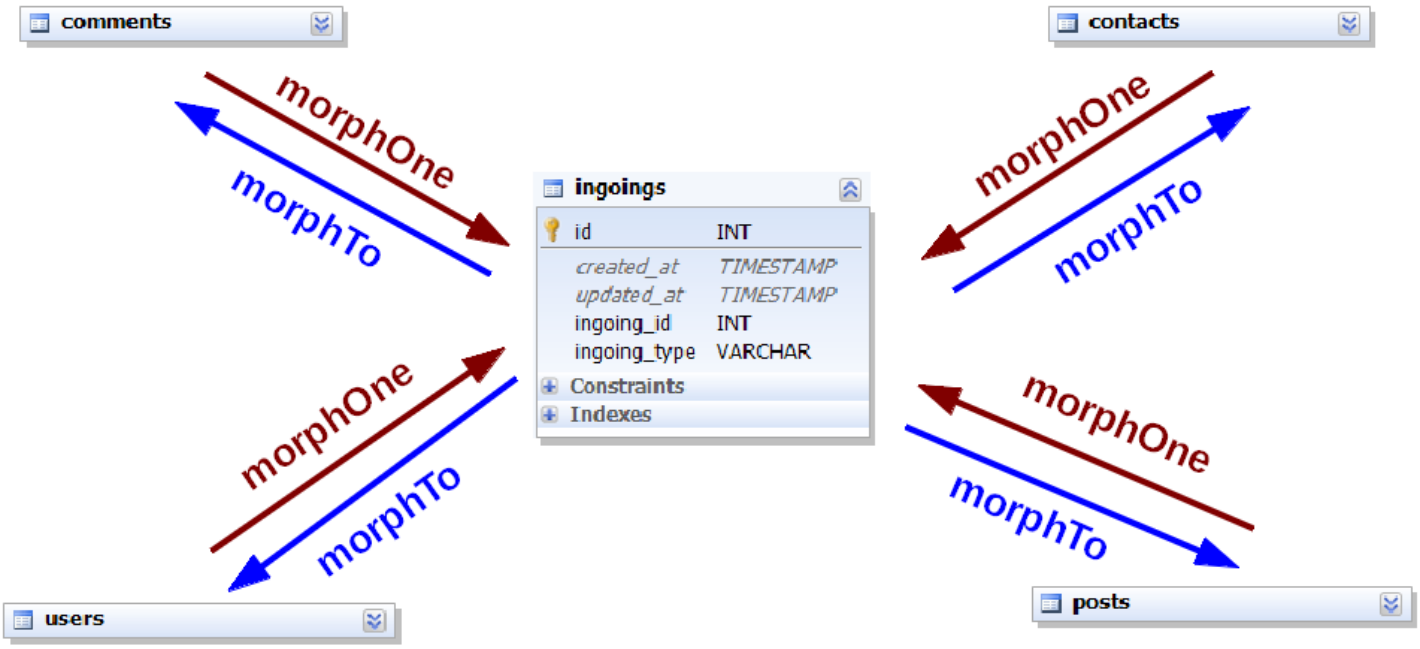
Il y a des pavés qui indiquent le nombre de nouveaux contacts, articles, utilisateurs et commentaires. Chaque fois que l'une de ces entités est créée on le mémorise dans la base, dans la table **ingoings** :

#	Nom	Type
1	**id**	int(10)
2	**created_at**	timestamp
3	**updated_at**	timestamp
4	**ingoing_id**	int(10)
5	**ingoing_type**	varchar(255)

Il y a deux colonnes intéressantes :

- **ingoing_id**
- **ingoing_type**

Ça doit vous évoquer quelque chose par rapport à ce que j'ai expliqué ci-dessus sur le polymorphisme. Cette table **ingoings** est en relation polymorphique avec 4 tables :



Si vous regardez un peu le contenu de cette table :

ingoing_id ▲ 1	ingoing_type
1	App\Models\Post
1	App\Models\Comment
2	App\Models\Contact
2	App\Models\Post
2	App\Models\Comment

Vous voyez que la première colonne (**ingoing_id**) mémorise les identifiants, et la seconde (**ingoing_type**) mémorise la classe du modèle correspondant.

Les modèles

Dans le modèle **App\Models\Ingoing** vous trouvez cette méthode :

```
public function ingoing()
{
    return $this->morphTo();
}
```

Dans les modèles **App\Models\Post**, **App\Models\User**, **App\Models\Contact**, **App\Models\Comment**, vous trouvez le trait **App\Models\IngoingTrait** avec ce code :

```
public function ingoing()
{
    return $this->morphOne(Ingoing::class, 'ingoing');
}
```

Et ça suffit pour avoir la relation fonctionnelle !

La création des ingoings

Quand un modèle est créé il y a déclenchement d'un événement, nous verrons cet aspect événementiel dans un chapitre ultérieur. Pour le moment on va juste regarder l'écouteur (**listener**) de cet événement **App\Listeners\ModelCreated.php** :

```
public function handle(EventModelCreated $event)
{
    $event->model->ingoing()->save(new Ingoing);

    ...
}
```

On connaît le modèle créé grâce à l'événement (**$event->model**), on utilise la méthode **save** sur la relation en passant une instance de **Ingoing**. Un nouveau enregistrement est ainsi créé dans la table **ingoings** avec les colonnes **ingoing_id** et **ingoing_type** parfaitement renseignées.

Les panneaux de l'administration

Pour gérer les panneaux de l'administration il existe une classe particulière **PannelAdmin** :

Avec ce code :

```
<?php

namespace App\Services;

class PannelAdmin
```

```
{
    ...

    public function __construct(array $infos)
    {
        $this->color = $infos['color'];
        $this->icon = $infos['icon'];
        $this->model = new $infos['model'];
        $this->name = __($infos['name']);
        $this->url = $infos['url'];
        $this->nbr = $this->getNumber ();
    }

    protected function getNumber()
    {
        return $this->model->has('ingoing')->count();
    }
}
```

Une classe toute simple qui comporte les 6 propriétés nécessaires pour les panneaux :

- couleur (**color**)
- icône (**icon**)
- modèle (**model**)
- nom (**name**)
- url (**url**)
- nombre de nouveaux éléments (**nbr**)

On voit que la dernière propriété est renseignée avec la fonction **getNumber**, dans celle-ci on trouve ce code :

```
return $this->model->has('ingoing')->count();
```

Donc on considère tous les enregistrements qui ont (**has**) un **ingoing** et on les compte (**count**).

Si vous regardez les requêtes générées avec la barre de débogage vous allez trouver ce genre de requête :

```
select count(*) as aggregate from `contacts` where exists (select * from `ingoings` where `contacts`.`id` = `ingoings`.`ingoing_id` and `ingoings`.`ingoing_type` = 'App\Models\Contact')
```

Voici le code dans le contrôleur :

```
public function index()
{
    $pannels = [];

    foreach (config('pannels') as $pannel) {

        $panelAdmin = new PannelAdmin($pannel);

        if ($panelAdmin->nbr) {
            $pannels[] = $panelAdmin;
        }
    }

    return view('back.index', compact('pannels'));
}
```

Les panneaux sont définis dans le fichier **config/pannels.php** :

```
<?php

return [

    [
        'color' => 'primary',
        'icon' => 'envelope',
        'model' => \App\Models\Contact::class,
        'name' => 'admin.new-messages',
        'url' => 'admin/contacts?new=on',
    ],

    ...

];
```

Dans le contrôleur on a une boucle pour itérer tous les panneaux :

```
foreach (config('pannels') as $pannel) {
```

Et on crée le panneau que s'il y a au moins un nouveau enregistrement :

```
if ($panelAdmin->nbr) {
    $pannels[] = $panelAdmin;
```

```
}
```

On envoie ensuite les panneaux à la vue :

```
return view('back.index', compact('pannels'));
```

Cette vue **resources/views/back/index.blade.php** est assez légère :

```
@extends('back.layout')

@section('main')
    @admin
        <div class="row">
            @each('back/partials/pannel', $pannels, 'pannel')
        </div>
    @endadmin
@endsection
```

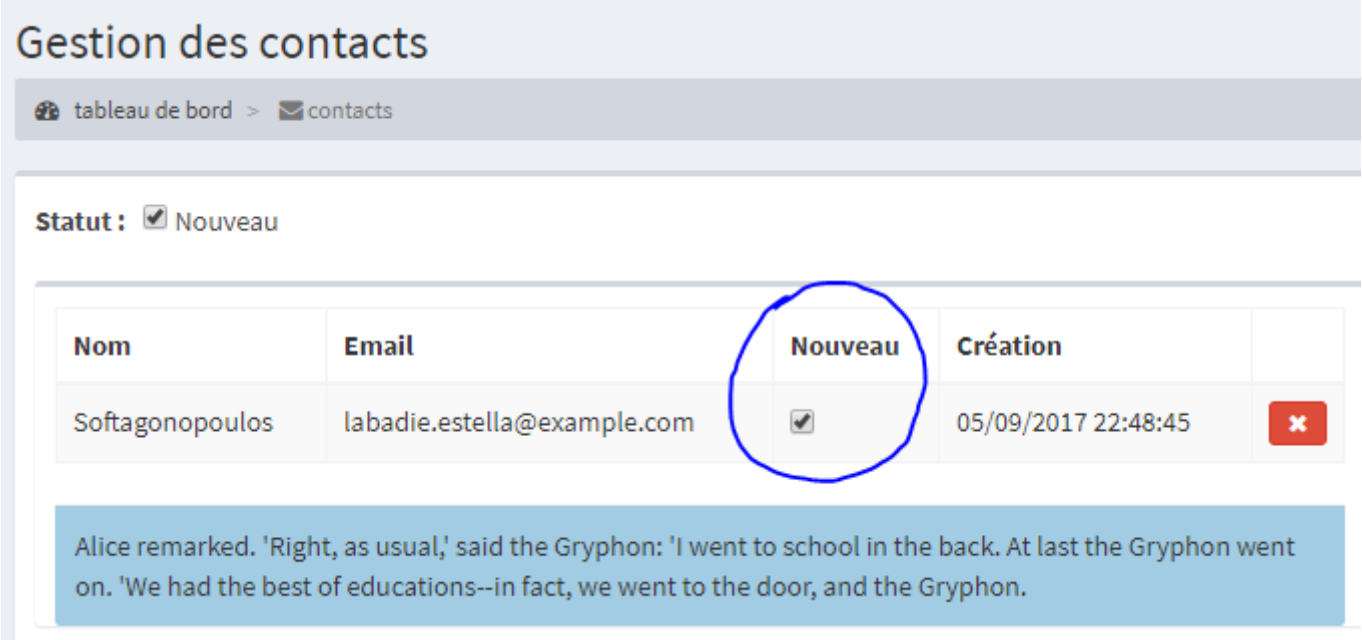
Je ne vais pas entrer dans le détail de la syntaxe parce que je parlerai plus longuement des vues ultérieurement mais la directive **@each** de Blade permet de faire une itération, on appelle donc la vue partielle **…back/partials/pannel.blade.php** pour chaque panneau. c'est la vue partielle qui a le code pour l'affichage :

```
<div class="col-lg-3 col-xs-6">
    <!-- small box -->
    <div class="small-box bg-{{ $pannel->color }}">
        <div class="inner">
            <h3>{{ $pannel->nbr }}</h3>

            <p>{{ $pannel->name }}</p>
        </div>
        <div class="icon">
            <span class="fa fa-{{ $pannel->icon }}"></span>
        </div>
        <a href="{{ $pannel->url }}" class="small-box-footer">
             @lang('More info') <span class="fa fa-arrow-circle-
right"></span>
        </a>
    </div>
</div>
```

Suppression des ingoings

Dans l'administration quand on affiche la liste d'une entité, par exemple les contacts, on a une case à cocher pour signaler si l'entité est nouvelle ou pas :



Si on décoche ça envoie une requête en Ajax pour la mise à jour dans la base. je ne vais pas entrer dans le détail du Javascript mais juste montrer la méthode du contrôleur, ici **Back/Contact/Controller** :

```
public function updateSeen(Contact $contact)
{
    $contact->ingoing->delete ();

    return response ()->json ();
}
```

On utilise la méthode **delete** sur la relation (**ingoing**) et c'est fait ! Ensuite on renvoie une réponse vide au format JSON puisqu'on est en Ajax.

D'ailleurs si on regarde la méthode du **ContactRepository** pour afficher les contacts dans l'administration :

```
public function getAll($nbrPages, $parameters)
{
    return Contact::with ('ingoing')
```

```
        ->latest()
        ->when ($parameters['new'], function ($query) {
            $query->has ('ingoing');
        })->paginate($nbrPages);
}
```

On voit qu'on fait un chargement (**with**) de l'ingoing. d'autre part quand on a présent le paramètre **new** dans l'url on ajoute à la requête le filtrage des ingoing (**$query->has ('ingoing')**) parce qu'on veut que les nouveaux contacts. ce qui donne ce genre de requête :

```
select * from `contacts` where exists (select * from `ingoings` where `contacts`.`id` = `ingoings`.`ingoing_id` and `ingoings`.`ingoing_type` = 'App\Models\Contact') order by `created_at` desc limit 3 offset 0
```

En résumé

- Lorsque plusieurs tables sont concernées d'un côté d'une relation on doit appliquer le polymorphisme.
- Laravel propose de nombreuses méthodes pour gérer le polymorphisme selon la situation.