Développer en back-end

ORM Eloquent – partie 2

Formatrice : Elidrissi Asmae

OFPPT
La Voie de l'Avenir

Plan du cour

Requêtes de récupération des données 01

Définition et récupération des données via les relations 02

Pagination de la base de données 03

Requêtes de récupération des données

Requêtes de récupération des données

- Le modèle Eloquent est considéré comme un puissant générateur de requêtes ;
- Il permet d'interroger d'une manière fluide la table de base de données associée au modèle;
- Le modèle Eloquent repose sur **Query Builder** de Laravel;
- On peut utiliser l'une des méthodes de Query Builder pour exprimer les requêtes sur un modèle;
- On peut ajouter des contraintes supplémentaires aux requêtes, puis invoquer la méthode get pour récupérer les résultats:

Exemples des requêtes de base:

```
$stagiaires=Stagiaire::all();
$stagiaire=Stagiaire::find(1);
```

Structure de requêtes:

```
$stagiaires= Stagiaire::where("note", 10)-> get();
```

Une ou plusieurs méthodes définissant la ou les clauses de la requête

Une méthode terminate pour récupérer les données

Requêtes de récupération des données

Afin de pouvoir tester, facilement, les différentes requêtes, on peut utiliser Tinker;

- Tinker est un REPL (boucle read-eval-print).
- REPL permet aux utilisateurs d'interagir avec l'application via la ligne de commande.
- Il est couramment utilisé pour l'interaction avec Eloquent ORM, les travaux, les événements, etc.
- C'est un outil qui permet d'entrer des expressions, de les faire évaluer et d'avoir le résultat à l'affichage
- Pour lancer cet outil, il suffit d'exécuter dans le terminal la commande suivante : **`php artisan tinker`**

Exemple :

```
$ php artisan tinker
Psy Shell v0.11.12 (PHP 8.2.0 — cli) by Justin Hileman
> use App\Models\Stagiaire
> Stagiaire::all()
= Illuminate\Database\Eloquent\Collection {#4685
    all: [
      App\Models\Stagiaire {#4687
        id: 1,
        nom_complet: "Tazi Naoual",
        genre: "F",
        date_naissance: "2002-03-12",
        note: "18.00",
        groupe_id: 1,
        created_at: "2023-02-20 08:02:47",
        updated_at: "2023-03-07 08:49:16",
        ville: "Fès",
```

Requêtes de récupération des données : Exemples

Pour les exemples des requêtes qui suiveront, on va exploiter le modèle « Stagiaire » avec lequel on a réalisé les exemples de cours précédents ;

Voici son Schéma de création :

```
Schema::create('stagiaires', function (Blueprint $table) {
            $table->id();
            $table->string("nom_complet", 300);
            $table->enum("genre", ["M", "F"])->default("M");
            $table->date("date_naissance")->nullable();
            $table->decimal("note", 4, 2);
             $table->string("ville", 50);
            $table->foreignId('groupe_id')->constrained();
            $table->timestamps();
        });
```

Requêtes de récupération des données : Exemples

- Les stagiaires qui ont eu une note **supérieure** à 10

```
$stagiaires= Stagiaire::where("note", ">=", "10")->get();
```

- Les stagiaires du groupe 1 **et** ayant le nom qui commence par « A »

```
$stagiaires= Stagiaire::where("groupe_id", 1)
                      ->where("nom_complet", "like", "A%")
                      ->get();
```

- Les stagiaires du groupe 1 **ou** qui ont un nom qui se termine par « A » ;

```
$stagiaires= Stagiaire::where("groupe_id", 1)
                       ->orwhere("nom_complet", "like", "%A")
                       ->get();
```

- Les stagiaires dont **on connais pas** la date de naissance

```
$stagiaires= Stagiaire::whereNull("date_naissance")
                       ->get();
```

- Les stagiaires qui ont eu une note **entre** 12 et 17

```
$stagiaires= Stagiaire::whereBetween("note", [12,17])
                              ->get();
```

- Les stagiaires inscrits au cours de l'année en cours :

```
Stagiaire::whereYear('created_at', date('Y'))->get()
```

*(On peut même utiliser: **whereDate()**, **whereMonth()**, **whereDay()**, **whereYear()** et **whereTime()**)*

Requêtes de récupération des données : Exemples

- Les stagiaires des groupes 1 et 2:

```
Stagiaire::whereIn('groupe_id', [1,2])->get()
```

- La liste des **différentes** villes des stagiaires:

```
Stagiaire::select("ville")->distinct()->get();
```

- Les noms et les notes des stagiaires sur 40 (La note multipliée par 2) : ***(Colonne calculée)***

```
Stagiaire::select("nom_complet")->selectRaw('note * 2 as note_sur_40')->get();
```

- Les noms et dates de naissance des stagiaires **triés** par date de naissance (Décroissant).

```
Stagiaire::select("nom_complet", "date_naissance")
        ->orderBy("date_naissance", "DESC")->get();
```

- La **meilleure** note du groupe numéro 1 ;

```
Stagiaire::where("groupe_id", 1)->orderBy("note", "DESC")->select("note")->first();
```

ou

```
Stagiaire::where("groupe_id", 1)->max("note")
```

- La **moyenne** des notes des stagiaires nés avant le 01/01/2003

```
Stagiaire::where("date_naissance","<", "2003-01-01")->avg("note");
```

Requêtes de récupération des données : Exemples

- La note **la plus basse** :

```
Stagiaire::min("note");
```

- Le **nombre** des stagiaires enregistrés dans la table stagiaires

```
Stagiaire::count()
```

- Le **nombre** de stagiaires **par groupe**

```
Stagiaire::selectRaw("groupe_id, count(*) as nombre_stagiaires")
        ->groupBy("groupe_id")->get();
```

- Les meilleures notes **par groupe**

```
Stagiaire::selectRaw("groupe_id, max(note) as meilleure_note")
        ->groupBy("groupe_id")->get();
```

- Les 3 meilleures stagiaires :

```
Stagiaire::orderBy("note", "DESC")->limit(3)->get(); //on utilise limit ou take
```

- Le(s) stagiaire(s) qui a(ont) eu la meilleure note :

```
Stagiaire::where("note", Stagiaire::max('note'))->get();
```

Requêtes de récupération des données : Exemples

- Les groupes ayant une moyenne des notes dépassant 11 :

```
Stagiaire::selectRaw("groupe_id, avg(note) as moyenne_notes")
        ->groupBy("groupe_id")
        ->having("moyenne_notes", ">", 11)
        ->get()
```

- Les groupes où sont inscrits entre 20 et 25 stagiaires:

```
Stagiaire::selectRaw("groupe_id, count(*) as nombre_stg")
        ->groupBy("groupe_id")
        ->havingBetween("nombre_stg", [20, 25])
        ->get()
```

Définition et récupération des données via les relations

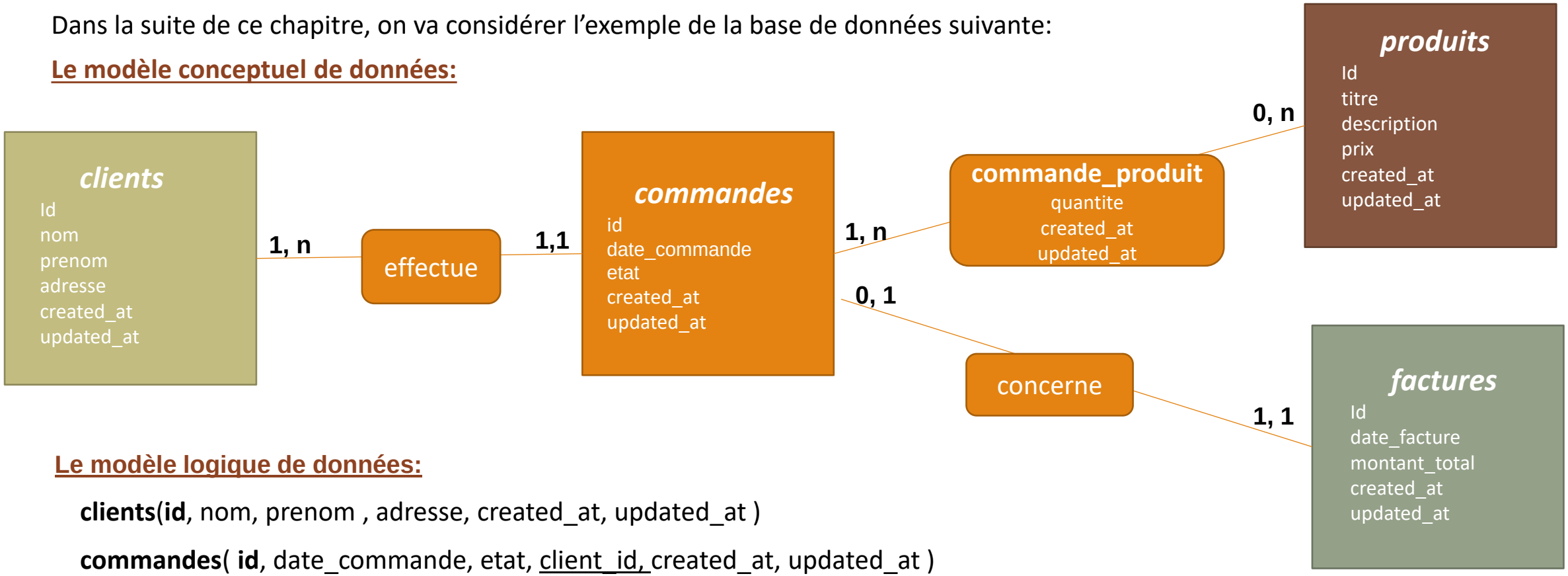
Définition et récupération des données via les relations

- Les relations dans Eloquent permettent de définir des liens entre des tables de bases de données pour simplifier la gestion de données .
- En utilisant les relations dans Eloquent, les développeurs peuvent éviter la duplication de code et réduire le temps nécessaire pour écrire des requêtes SQL complexes.
- Les relations Eloquent sont définies comme des méthodes sur les classes de modèle Eloquent.
- Les relations servent également de puissants générateurs de requêtes ;
- Il existe plusieurs types de relations possibles entre les modèles de données:
  - **hasMany()** : pour les relations 1 vers 0..* (de un à plusieurs)
  - **belongsTo()** : pour les relations 0..* vers 1 ou 0..1 vers 1
  - **belongsToMany()** : pour les relations 0..* vers 0..* (avec une table pivot)
  - **hasOne()** : pour les relations 1 vers 0..1 (de un à un)
  - **hasManyThrough()** : pour les relations 1 vers 0..* suivies d'une autre de 1 vers 0..*

Définition et récupération des données via les relations : Exemple de cours

Dans la suite de ce chapitre, on va considérer l'exemple de la base de données suivante:

Le modèle conceptuel de données:



Le modèle logique de données:

clients(**id**, nom, prenom , adresse, created_at, updated_at)

commandes(**id**, date_commande, etat, client_id, created_at, updated_at)

produits(**id**, titre, description, prix, created_at, updated_at)

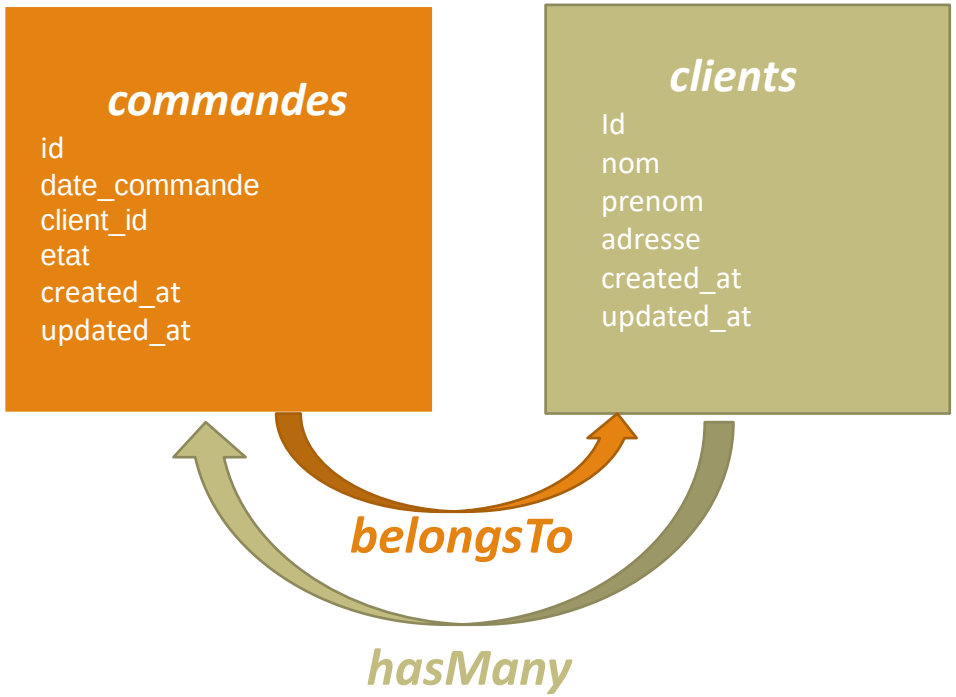
commande_produit(**commande_id, produit_id** , quantite, created_at, updated_at)

Factures(**id**, date_facture, montant_total, commande_id, created_at, updated_at)

Notez: Avant de procéder à la définition des relations entre les modèles il faut, tout d'abord, créer cette structure de tables dans la base de données (Exécuter les 5 migrations)

Les relations hasMany() et belongsTo

- Dans une relation **hasMany** (**un à plusieurs**), chaque enregistrement dans la table parent peut être associé à plusieurs enregistrements dans la table enfant.
- Dans le contexte de notre base de données, chaque client peut avoir **plusieurs** commandes, mais chaque commande ne peut être associée qu'à un seul client. Cela signifie qu'un client peut passer plusieurs commandes, mais chaque commande ne peut être passée que par un seul client.
- Avec les relations Eloquent, cela se traduit par:
  - « **clients hasMany Commandes**».
  - « **commandes belongsTo Clients** ».



Les relations hasMany() et belongsTo

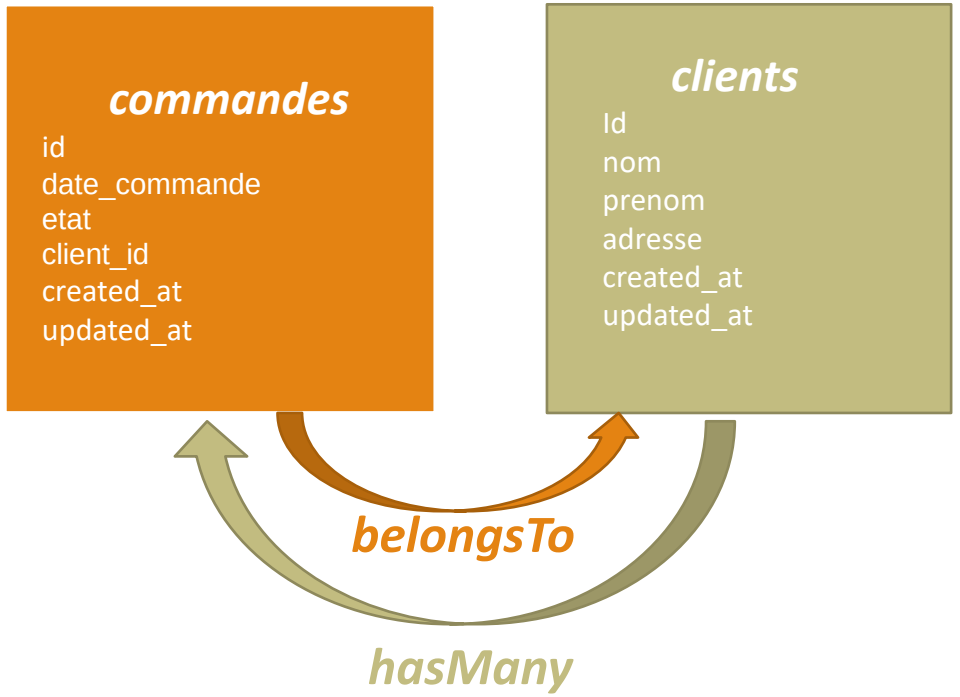
Définition des relations :

Pour définir les relations, on ajoute dans le modèle **Client** la méthode :

```
public function commandes() {
    return $this->hasMany(Commande::class);
}
```

Et dans le modèle **Commande** la méthode :

```
public function client() {
    return $this->belongsTo(Client::class);
}
```



L'ajout via les relations

Pour insérer des modèles « commandes » via les relations prédéfinies, on peut écrire:

```
$client->commandes()->save($commande1)
```

ou

```
$client->commandes()->saveMany([$commande1, $commande2 ]);
```

Les relations hasMany() et belongsTo

Interroger les relations :

Tous les types de relations Eloquent servent également de générateurs de requêtes , Elle permettent de continuer à enchaîner les contraintes sur la requête de relation avant d'exécuter finalement la requête SQL sur votre base de donnée.

Exemples d'utilisation des relations en tant que méthodes :

- **Liste des commandes du client numéro 2 passé durant l'année en cours:**

```
$client=Client::find(2);
$client->commandes()
        ->whereYear("date_commande", date("Y"))
        ->get();
```

//Requête SQL : select * from `commandes` where `commandes`.`client_id` = ? and `commandes`.` client_id` is not null and year(`date_commande`) = ?

- **Liste des commandes du client numéro 2 passé durant l'année en cours ou mis à jour durant l'année en cours:**

```
use Illuminate\Database\Eloquent\Builder;
$client=Client::find(2);
$client->commandes()
        ->where(function(Builder $query){
                $query->whereYear("date_commande", date("Y"))
                      ->orWhereYear("updated_at", date("Y"));
        })->get();
```

//Requête SQL : select * from `commandes` where `commandes`.`client_id` = ? and `commandes`.`client_id` is not null and (year(`date_commande`) = ? or year(`updated_at`) = ?)

Les relations hasMany() et belongsTo

Interroger les relations :

Exemples d'utilisation des relations en tant que propriété :

- **Liste des commandes d'un client donné:**

```
$client=Client::find($id);
$commandes=$client->commandes;
```

- **Le nombre des commandes d'un client donné:**

```
$client=Client::find($id);
$commandes=$client->commandes->count();
```

- **Le nom et prénom du client qui a effectué une commande donné:**

```
$commande = Commande::find($id);
$nom_complet = $commande->client->nom." ". $commande->client->prenom;
```

Les relations hasMany() et belongsTo

Interroger les relations : Exemples de méthodes interrogeant les relations :

- **Liste des commandes des clients 1, 2 et 3:**

```
$clients=Client::find([1, 2, 3]);
$commandes=Commande::whereBelongsTo($clients)->get();
```

//Requête SQL : "select * from `commandes` where `commandes`.`client_id` in (?, ?, ?)

- **Liste des clients qui ont passé au moins une commande:**

```
Client::has("commandes")->get();
```

//Requête SQL : "select * from `clients` where exists (select * from `commandes` where `clients`.`id` = `commandes`.`client_id`)“

- **Liste des clients qui ont passé au moins 2 commande:**

```
Client::has("commandes", ">=", 2)->get();
```

//Requête SQL : select * from `clients` where (select count(*) from `commandes` where `clients`.`id` = `commandes`.`client_id`) >= 2

- **Liste des clients qui ont passé des commandes avec facture :**

(on suppose qu’on a déjà défini la relation commande-facure):

```
Client::has("commandes.facture)->get();
```

//Requête SQL : select * from `clients` where exists (select * from `commandes` where `clients`.`id` = `commandes`.`client_id` and exists (select * from `factures` where `commandes`.`id` = `factures`.`commande_id`))

Les relations hasMany() et belongsTo

Interroger les relations : Exemples de méthodes interrogeant les relations :

- **Liste des clients qui ont passé au moins une commande cette année:**

```
Client::whereHas("commandes", function(Builder $query){
        $query->whereYear("date_commande",date('Y'));
        })->get();
```

//Requête SQL : select * from `clients` where exists (select * from `commandes` where `clients`.`id` = `commandes`.`client_id` and year(`date_commande`) = ?)

- **Liste des clients qui ont passé au moins DEUX commandes cette année:**

```
Client::whereHas("commandes", function(Builder $query){
        $query->whereYear("date_commande",date('Y'));
        }, ">=", 2)->get();
```

//Requête SQL : select * from `clients` where (select count(*) from `commandes` where `clients`.`id` = `commandes`.`client_id` and year(`date_commande`) = ?) >= 2

- **Liste des clients qui ont confirmé leurs commandes:**

```
Client::whereRelation("commandes", "etat", "confirmée")->get();
```

//Requête SQL : select * from `clients` where exists (select * from `commandes` where `clients`.`id` = `commandes`.`client_id` and `etat` = ?)

Les relations hasMany() et belongsTo

Interroger les relations : Exemples de méthodes interrogeant les relations :

- **Le nombre de commandes par client:**

```
$resultats=Client::withCount("commandes")->get();

  * $resultat sera un tableau comportant, en plus des propriétés du modèle Client, une
  nouvelle propriété appelée commandes_count
```

//Requête SQL : select `clients`.*, (select count(*) from `commandes` where `clients`.`id` = `commandes`.`client_id`) as `commandes_count` from `clients`

- **Le nombre de commandes passées avant le 01/01/2023 par client:**

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

$res= Client::withCount(["commandes" => function(Builder $query){
        $query->where("date_commande", "<", "2023-01-01");
    }])->get();
```

//Requête SQL : select `clients`.*, (select count(*) from `commandes` where `clients`.`id` = `commandes`.`client_id` and `date_commande` < ?) as `commandes_count` from `clients`

- **Le nombre de commandes passées avant le 01/01/2023 par un client (objet) donné:**

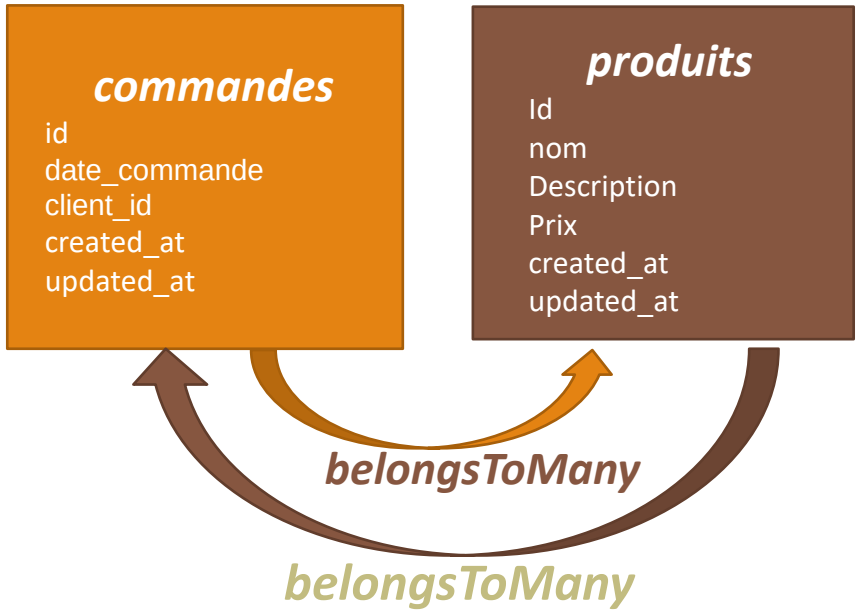
```
$client=Client::find($id);
$res= $client->loadCount("commandes");
```

⇒ *En plus de la méthode **withCount**, Eloquent fournit les méthodes **withSum**, **withMin**, **withMax**, **withAvg** , **loadSum, loadMin, loadMax, loadAvg:***

⇒ ***Ex. :*** `Client::withMax("commandes as dernière_commande", "date_commande")->get();`

La relation many-to-many : belongsToMany()

- En Laravel ORM Eloquent, la relation **belongsToMany** (**appartient à plusieurs**) est utilisée pour établir une relation de type **many-to-many** (plusieurs-à-plusieurs) entre deux tables de la base de données.
- Cette relation est utilisée lorsque chaque enregistrement dans une table peut être associé à plusieurs enregistrements dans une autre table, et vice versa.
- Par exemple, une table « ***commandes*** » peut avoir plusieurs « ***produits*** », et chaque produit peut être associé à plusieurs commandes.
- Le passage du MCD au MLD a donné la naissance d'une nouvelle table « ***pivot*** » comportant deux clés étrangères.



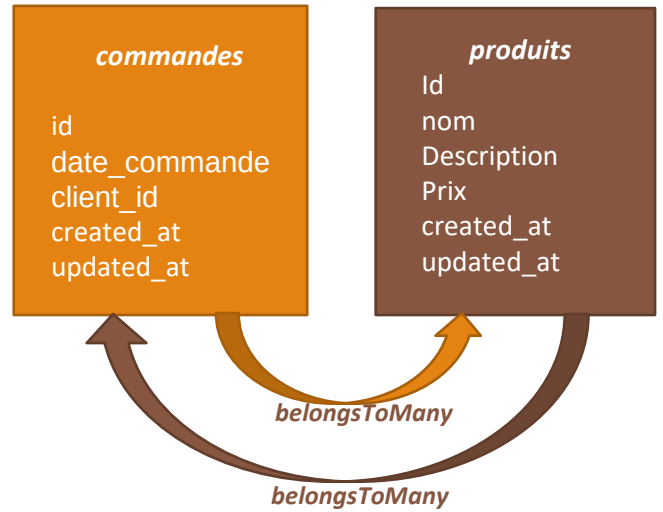
La relation many-to-many : belongsToMany()

- En Eloquent, la relation ***many-to-many*** entre les tables "commandes" et "produits" peut être représentée en utilisant **une table pivot**, qui contient les **clés étrangères** des enregistrements de chaque table associée.
- Cette table pivot permet de relier les enregistrements des deux tables en créant une association many-to-many entre eux.
- Laravel, génère les noms des tables pivots en concaténant les noms des deux tables et en les triant selon leur nom
- Il y a, toujours, la possibilité de personnaliser les noms des tables pivots.
- Pour notre exemple, on a suivi la convention Laravel, et on a appelé la table pivot : **commande_produit**

Définition des relations:

- Voici comment vous pouvez définir la relation many-to-many entre les modèles "Commande" et "Produit" en utilisant Laravel ORM Eloquent :
- Dans le modèle "**Commande**", ajoutez la méthode suivante :

```
public function produits()
{
    return $this->belongsToMany(Produit::class);
}
```



- Dans le modèle "**Produit**", ajoutez la méthode suivante :

```
public function commandes() {
 return $this->belongsToMany(Commande::class);
}
```

La relation many-to-many : belongsToMany()

⇒**Peronnaliser le nom de la table pivot:**

```
class Commande extends Model
{
    public function produits(){
        return $this->belongsToMany(Produit::class,"ligne_commande");
    }
}
```

⇒Ce code indique que la table pivot prendera le nom “ligne_commande”

⇒**Peronnaliser les noms des clés étrangères de la table pivot**

```
class Commande extends Model
   {
      public function produits(){
           return $this->belongsToMany(Produit::class,"ligne_commande“, “clé_commande”,
   “clé_produit”);
       }
   }
```

La relation many-to-many : belongsToMany()

⇒ **Ajouter les champs : created_at et updated_at à la table pivot:**

```
class Commande extends Model
   {
       public function produits(){
           return $this->belongsToMany(Produit::class)
             ->withTimestamps();
       }
   }
```

⇒ **Ajouter une (ou plusieurs) colonnes à la table pivot:**

```
class Commande extends Model
{
    public function produits(){
        return $this->belongsToMany(Produit::class,"ligne_commande")
               ->withPivot("quantité"); //si on veut ajouter deux colonnes: quantité et validé,
                                                    on écrit withPivot("quantité", "validé")     }
}
```

Notez:

- *Toutes ces configurations doivent être définies sur les deux classes modèles: Commande et Produit*
- *Il faut mettre à jour la structure de table pivot dans la base de données pour qu'elle soit synchronisée avec les nouvelles configurations*

La relation many-to-many : belongsToMany()

Interroger les relations :

Après avoir défini cette relation, nous pouvons accéder à la table intermédiaire en utilisant l'attribut **pivot** sur les modèles :

Exemples:

- **Les quantités des produits de la commande numéro 3 :**

```
$commande = Commande::find(3);
foreach ($commande->produits as $prd) {
    echo $prd->pivot->quantite;
}
```

- **Sous blade on suppose qu'on a reçu un objet $commande et on veut afficher son détail: les titres et quantités des produits commandées:**

```
@foreach($commande->produits as $produit)
   <p>{{$produit >titre}}
      ({{$produit->pivot->quantite}})</p>
@endforeach
```

- **Les produits qui ont été commandé avec une quantité =1 de la commande numéro 3:**

```
$commande = Commande::find(3);
$commande->produits()->wherePivot("quantite", 1)
->get();
```

```
. \
. $commande = Commande::find(3);\
. $commande->produits()->wherePivot("quantite", 1)\
. ->get();
= Illuminate\Database\Eloquent\Collection {#7290
    all: [
      App\Models\Produit {#7271
        id: 2,
        titre: "salle à manger",
        description: "Table ronde 120cm-80cm avec 4 chaises",
        prix: "4500.00",
        created_at: null,
        updated_at: null,
        pivot: Illuminate\Database\Eloquent\Relations\Pivot {#7182
          commande_id: 3,
          produit_id: 2,
          quantite: 1,
          created_at: null,
          updated_at: null,
        },
      },
      App\Models\Produit {#7288
        id: 3,
        titre: "Lit double",
        description: "160x200 super qualité",
        prix: "4500.00",
        created_at: null,
        updated_at: null,
        pivot: Illuminate\Database\Eloquent\Relations\Pivot {#7215
          commande_id: 3,
          produit_id: 3,
          quantite: 1,
          created_at: null,
          updated_at: null,
        },
      },
    ],
  }
```

La relation many-to-many : belongsToMany()

Interroger les relations :

- **Le montant total à payer pour la commande numéro 2:**

```
$cmd=Commande::find(1);
$cmd->produits->sum(function($prd){
    return $prd->prix*$prd->pivot->quantite;
});
```

- **Liste des commandes où on a commandé plus de 3 produits:**

```
Commande::has("produits", ">=", 3)->get();
```

- **Liste des commandes où on a commandé des produits avec des prix compris entre 5000 et 10000:**

```
Commande::whereHas("produits", function(Builder $query){
    $query->whereBetween("prix",[5000, 10000]);
})->get();
```

La relation many-to-many : belongsToMany()

Associer des modèles :

- La méthode "attach()" vous permet d'ajouter une relation entre un enregistrement de la table "**commande**" et un enregistrement de la table "**produit**".
- Par exemple, si vous souhaitez ajouter un produit ayant l'identifiant **1** à une commande ayant l'identifiant **2**, vous pouvez utiliser la méthode suivante :

```
$commande = Commande::find(2);
$commande->produits()->attach(1);
```

Cette méthode ajoute une nouvelle ligne à la table pivot contenant les clés étrangères correspondantes pour la commande et le produit.

Synchroniser les relations :

- La méthode "**sync**()" vous permet de synchroniser les relations entre une commande et une liste de produits donnée.
- Elle supprime les relations existantes et crée de nouvelles relations pour les produits spécifiés.
- Par exemple, si vous souhaitez synchroniser les produits ayant les identifiants 1, 2 et 3 avec la commande ayant l'identifiant 2, vous pouvez utiliser la méthode suivante :

```
$commande = Commande::find(2);
$commande->produits()->sync([1, 2, 3]);
```

Cette méthode supprime toutes les relations existantes entre la commande et les produits et crée de nouvelles relations pour les produits spécifiés.

La relation many-to-many : belongsToMany()

Détacher les modèles :

- La méthode "***detach***()" supprime une relation entre un enregistrement de la table "commande" et un enregistrement de la table "produit".
- Par exemple, si vous souhaitez supprimer la relation entre un produit ayant l'identifiant 1 et une commande ayant l'identifiant 2, vous pouvez utiliser la méthode suivante :

```
$commande = Commande::find(2);
$commande->produits()->detach(1);
```

- Cette méthode supprime la ligne correspondante dans la table pivot pour la commande et le produit spécifiés.
- Si vous souhaitez supprimer toutes les relations entre une commande et ses produits, vous pouvez utiliser la méthode "detach()" sans argument

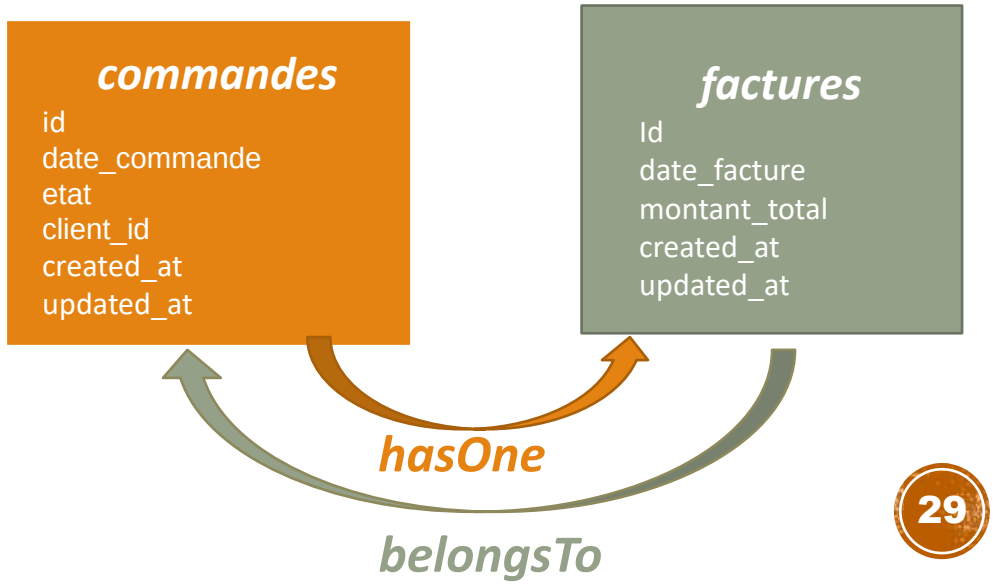
```
$commande = Commande::find(2);
$commande->produits()->detach();
```

La relation one to one: hasOne

- On entend par relation **1,1** une relation dont les cardinalités **maximales sont à 1** de chaque côté
- Exemples: Pays ⇔ Capital // Personne ⇔ empruntes
- Lors de la définition de la base de données : la clé primaire de la table la plus importante (fonctionnellement) migre vers la seconde table;
- La relation **hasOne** en Laravel ORM Eloquent permet de définir une relation "un-à-un" entre deux tables, où chaque enregistrement de la première table (par exemple "Commande") est associé à **un enregistrement unique** dans la seconde table (par exemple "Facture").
- Voici un exemple de définition de la relation hasOne entre les tables "Commande" et "Facture" :

Modèle "Commande"

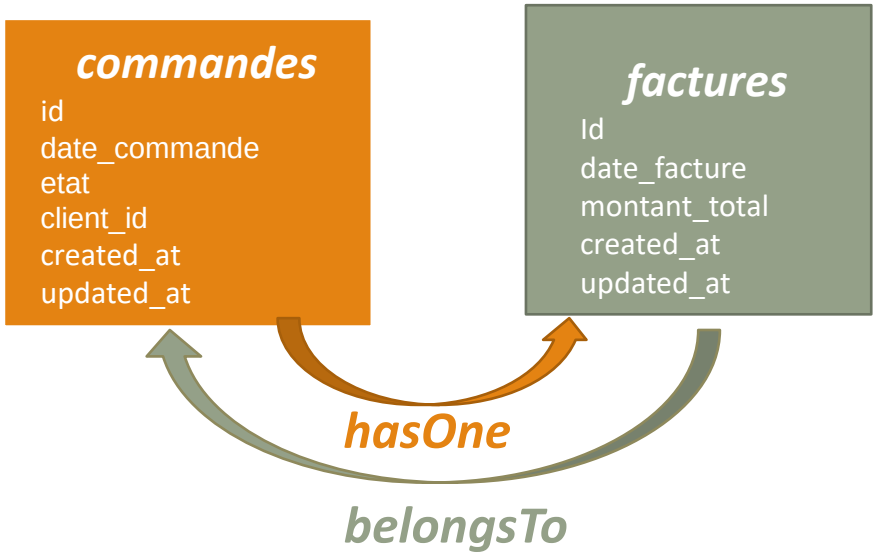
```
class Commande extends Model
{
    public function facture(){
            return $this->hasOne(Facture::class);
        }
}
```



La relation one to one: hasOne

Modèle "Facture"

```
class Facture extends Model
{
public function commande(){
        return $this->belongsTo(Commande::class);
    }
}
```



- On peut utiliser cette relation pour récupérer la facture associée à une commande donnée :

```
$commande = Commande::find(1);

$facture = $commande->facture;
```

Notez:

- Il est important de noter que la relation hasOne nécessite **une clé étrangère** dans la table correspondante, qui pointe vers la clé primaire de l'autre table.
- Dans l'exemple ci-dessus, la table "Facture" doit contenir une clé étrangère **"commande_id"** qui fait référence à la clé primaire "id" de la table "Commande".

Relation hasManyThrough

- La relation **hasManyThrough** en Laravel ORM Eloquent permet de définir une relation **"un-à-plusieurs-à-un**" entre trois tables, où chaque enregistrement de la première table (par exemple "Client") est associé à plusieurs enregistrements dans une deuxième table (par exemple « Commande"), qui sont eux-mêmes associés à des enregistrements dans une troisième table (par exemple "Facture").

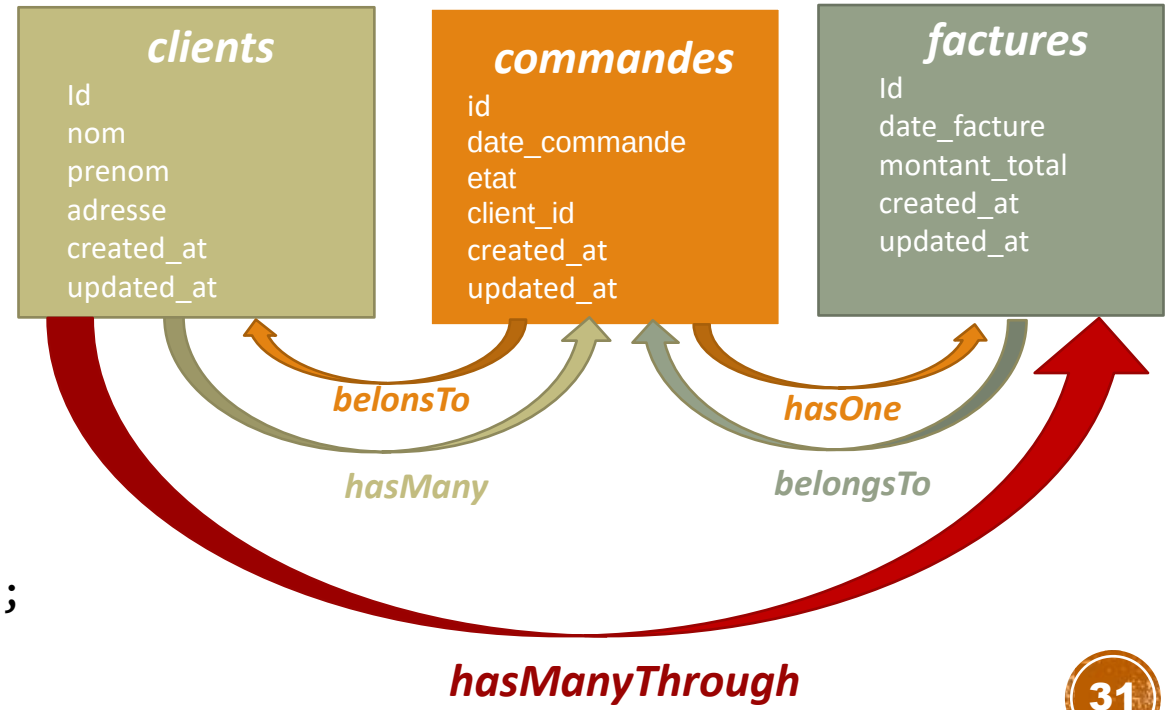
- Voici un exemple de définition de la relation **hasManyThrough** entre les tables "Client", "Commande" et " Facture" :

- Après avoir défini les différentes relations entre les trois tables, on peut en définir celle de « **hasManyThrough** » sous le modèle « **Client** »:

Modèle "Client“

```
class Client extends Model
{
    public function factures() {

      return $this->hasManyThrough(Facture::class,
                                   Commande::class);
    }
}
```



Relation hasManyThrough

- Dans cet exemple, la méthode "**factures**()" du modèle "**Client**" renvoie la relation "**hasManyThrough**" avec le modèle "**Facture**", à travers la table intermédiaire "**Commande**".
- Vous pouvez alors utiliser cette relation pour récupérer toutes les factures associés à un client donné :

```
$client=Client::find(2);
$factures=$client->factures;
```

- la relation **hasManyThrough** peut être utilisée pour représenter des relations **hasMany – hasOne ou hasMany – hasMany,** en fonction des modèles impliqués dans la relation.
- Dans l'exemple précédent, on l'a défini pour représenter une relation **hasMany - HasOne**;
- **Exemple de hasMany-hasMany** : Un utilisateur peut poster plusieurs publications et une publication peut avoir plusieurs commentaires, Ainsi, on peut définir la relation **hasManyThrough** sous le modèle « Utilisateur » de la façon suivante:

```
class Utilisateur extends Model
{
    public function commentaires()
    {
        return $this->hasManyThrough(Commentaire::class, Publication::class);
    }
}
```

Pagination de la base de données

Pagination de la base de données

- La pagination consiste à limiter le nombre de données affichées et de prévoir des boutons de navigation entre les pages ainsi constituées. Laravel propose une pagination automatique facile à mettre en œuvre aussi bien au niveau du Query Builder que d'Eloquent.

Configurer la pagination:

- On peut « paginer » le résultat d'une requête en utilisant la méthode « **paginate** »:

Exemples:

```
1. Commande::paginate(5);
2. Commande::where("etat", "confirmé")->paginate(3);
```

- L'argument passé en paramètre de la méthode « paginate » indique le nombre d'enregistrements qu'on souhaite afficher par page.

- La méthode ***paginate*** prend automatiquement en charge la configuration de la "limite" (limit) et du "décalage" (offset) de la requête en fonction de la page actuelle consultée par l'utilisateur.
- Par défaut, la page en cours est détectée par la valeur de l'argument ***page*** passé dans la requête HTTP. Cette valeur est automatiquement détectée par Laravel, et elle est automatiquement insérée dans les liens générés par le paginateur.
- La méthode « paginate » permet également de compter le nombre total des enregistrements retournés par la requête; Si on ne souhaite pas calculer ce nombre total, il suffit d'appeler la méthode ***simplePaginate.***

Pagination de la base de données

Afficher les résultats de la pagination:

- Par défaut, le HTML généré par le paginateur est compatible avec le framework CSS Tailwind ; cependant, la prise en charge de la pagination Bootstrap est également disponible.
- Exemple:

  - Si on souhaite afficher trois produits par page, on peut écrire dans la méthode ***index*** du contrôleur ***ProduitController***:

```
public function index()
{
    $produits=Produit::paginate(3);
    return view("produit.index", compact("produits"));
}
```

  - Sous blade, dans une vue sous « ***views/produit/index.blade.php*** », on ajoute le code suivant:

```
<table>
    @foreach($produits as $prd)
        <tr>
            <td>{{ $prd->titre }} </td>
            <td>{{ $prd->description }} </td>
            <td>{{ $prd->prix }} </td>
        </tr>
    @endforeach
</table>
{{ $produits->links(); }}
```

Pagination de la base de données

Afficher les résultats de la pagination:

```
{{ $produits->links(); }}
```

- La méthode ***links*** générera les liens vers le reste des pages.

- Laravel inclut des vues de pagination construites à l'aide de Bootstrap CSS. Pour utiliser ces vues au lieu des vues par défaut de Tailwind, vous pouvez appeler les méthodes ***useBootstrapFour*** ou ***useBootstrapFive*** de la classe ***Paginator*** dans la méthode ***boot*** de la classe **App\Providers\AppServiceProvider** :

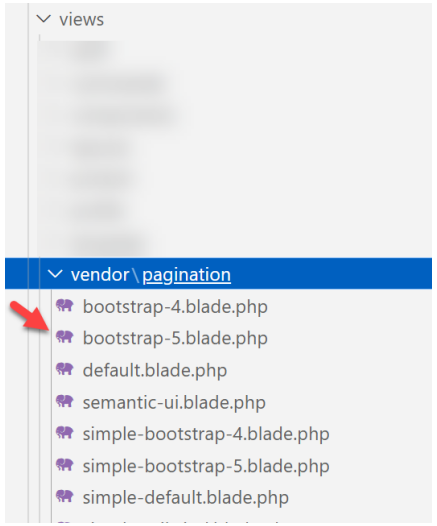
```
use Illuminate\Pagination\Paginator;
    public function boot(): void
    {
        Paginator::useBootstrapFive();
    }
```

Personnaliser les vues:

Si on souhaite personnaliser les vues de pagination, il suffit de les exporter dans le répertoire **resources/views/vendor** à l'aide de la commande **vendor:publish** :

```
php artisan vendor:publish --tag=laravel-pagination
```

Dans notre cas, on a appelé la méthode « paginate » et on a utilisé Bootstrap 5, il suffit de manipuler le fichier« bootstrap-5.blade.php » afin de personnaliser la vue de nos paginations;



TP

À la prochaine !