Eloquent ORM

Introduction

L'ORM Eloquent inclus avec Laravel fournit une implémentation magnifique et simple d'ActiveRecord pour travailler avec votre base de données. Chaque table de votre base de données a un Modèle associé pour interagir avec cette table.

Avant de commencer, n'oubliez pas de configurer votre connexion à la base de données dans le fichier `app/config/database.php`.

Utilisation basique

Pour commencer, créez un modèle Eloquent. Ils sont généralement stockés dans le dossier `app/models`, mais vous êtes libre de les mettre dans n'importe quel endroit qui peut être chargé automatiquement en accord avec votre fichier `composer.json`.

Définition d'un modèle Eloquent

```
class User extends Eloquent {}
```

Notez que nous n'avons pas indiqué à Eloquent quelle table doit être utilisée pour notre modèle `User`. Le nom de la classe en minuscule et au pluriel sera utilisé en tant que table à moins que vous ne définissiez une autre table explicitement. Donc dans ce cas, Eloquent utilisera le table `users` pour le modèle `User`. Pour définir explicitement un nom de table, définissez une propriété `$table` dans votre modèle :

```
class User extends Eloquent {

    protected $table = 'my_users';

}
```

Eloquent va également présumer que votre table a une clé primaire nommée `id`. Vous pouvez définir une propriété `primaryKey` pour surcharger cette convention. De la même manière, vous pouvez définir une propriété `connection` pour surcharger le nom de la connexion qui sera utilisé pour accéder à la table de ce modèle.

Une fois qu'un modèle est défini, vous êtes prêt à récupérer et à créer des enregistrements dans votre table. Notez que vous aurez besoin de créer des colonnes `updated_at` et `created_at` sur votre table par défaut. Si vous ne voulez pas de ces colonnes, qui sont auto-maintenues par Laravel, définissez une propriété `$timestamps` à `false`.

Retourne tous les modèles

```
$users = User::all();
```

Retourne un modèle par sa clé primaire

```
$user = User::find(1);

var_dump($user->name);
```

> **Note:** Toutes les méthodes disponibles dans le Query Builder sont également disponibles avec Eloquent.

Récupérer un modèle par sa clé primaire ou lancer une exception

Parfois vous pourriez vouloir lancer une exception si un modèle n'est pas trouvé, vous permettant d'attraper les exceptions en utilisant un gestionnaire d'événement `error` et afficher une page 404.

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

Pour enregistrer le gestionnaire d'erreur, écoutez auprès d'un `ModelNotFoundException`

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
});
```

Requêtage utilisant le modèle Eloquent

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user) {
    var_dump($user->name);
}
```

Bien sûr, vous pouvez également utiliser les fonctions d'agrégat du Query Builder.

Agrégat avec Eloquent

```
$count = User::where('votes', '>', 100)->count();
```

Si vous êtes dans l'impossibilité de générer la requête que vous souhaitez via l'interface Fluent, alors n'hésitez pas à utiliser la méthode `whereRaw` :

```
$users = User::whereRaw('age > ? and votes = 100', array(25))->get();
```

Ségmentation des résultats

Si vous avez besoin de traiter beaucoup (des milliers) d'enregistrements Eloquent, utiliser la méthode `chunk` vous permettra d'économiser beaucoup de RAM :

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

Le premier argument est le nombre de lignes que vous souhaitez recevoir par ségment, La fonction anonyme passée en second argument sera appelée pour chaque ségment provenant de la base de données.

Spécification de la connexion

Vous pouvez spécifier quelle connexion à la base de données est utilisée quand vous lancez une requête Eloquent. Utilisez simplement la méthode `on` :

```
$user = User::on('connection-name')->find(1);
```

Assignement de masse

Lorsque vous créez un nouveau modèle, vous passez un tableau d'attributs au constructeur du modèle. Ces attributs sont ensuite assignés au modèle via l'assignement de masse. C'est très pratique, cependant cela peut être un risque **sérieux** de sécurité lorsque des données provenant d'utilisateurs sont aveuglément passées dans un modèle. Dans ce cas, l'utilisateur est libre de modifier n'importe quel attribut du modèle.

Pour commencer, définissez les propriétés `fillable` ou `guarded` sur votre modèle.

La propriété `fillable` spécifie quels attributs peuvent être assignés en masse. Cela peut être défini dans la classe ou au niveau de l'instance du modèle.

Définition de l'attribut fillable dans un modèle

```
class User extends Eloquent {

    protected $fillable = array('first_name', 'last_name', 'email');

}
```

Dans cet exemple, seuls les trois attributs listés peuvent être assignés lors d'un assignement de masse.

L'inverse de `fillable` est `guarded` , et il contient une "blacklist" plutôt qu'un laisser passer :

Définition de l'attribut guarded dans un modèle

```
class User extends Eloquent {

    protected $guarded = array('id', 'password');

}
```

Dans l'exemple ci-dessus, les attributs `id` et `password` **ne peuvent pas** être assignés en masse. Tous les autres attributs peuvent être assignés lors d'un assignement de masse. Vous pouvez aussi bloquer **tous** les attributs lors de l'assignement de masse en utilisant guard :

Bloque tous les attributs lors de l'assignement de masse

```
protected $guarded = array('*');
```

Insertion, mise à jour, suppression

Pour créer un nouvel enregistrement dans la base de données pour un modèle, créez simplement une nouvelle instance d'un modèle et appelez la méthode `save` .

Sauvegarde un nouveau modèle

```
$user = new User;

$user->name = 'John';

$user->save();
```

> **Note:** Typiquement, votre modèle Eloquent aura une clé de type auto-increment. Cependant, si vous souhaitez spécifier votre propre clé, définissez la propriété `incrementing` de votre modèle à `false` .

Vous pouvez également utiliser la méthode `create` pour sauvegarder un modèle en une seule ligne. L'instance du modèle insér**é** sera retournée par la méthode. Cependant avant de faire cela, vous devrez spécifier l'attribut `fillable` ou `guarded` sur le modèle, car tous les modèles Eloquent sont protégés contre l'assignement de masse.

Après avoir sauvé ou créé un nouveau modèle qui utilise un ID auto-incrémental, vous pouvez retrouver l'ID en accédant à l'attribut `id` de l'objet.

Mise en place de l'attribut guarded sur le modèle

```
class User extends Eloquent {

    protected $guarded = array('id', 'account_id');

}
```

Création d'un utilisateur en utilisant la méthode create

```
// Create a new user in the database...
$user = User::create(array('name' => 'John'));

// Retrieve the user by the attributes, or create it if it doesn't exist...
$user = User::firstOrCreate(array('name' => 'John'));

// Retrieve the user by the attributes, or instantiate a new instance...
$user = User::firstOrNew(array('name' => 'John'));
```

Pour mettre à jour un modèle, récupérez-le, changez un attribut, et utilisez la méthode `save` :

Mise à jour d'un Modèle

```
$user = User::find(1);

$user->email = 'john@foo.com';

$user->save();
```

Parfois vous pourriez vouloir sauvegarder non seulement le modèle, mais aussi toutes ses relations. Pour ce faire, utilisez la méthode `push` :

Sauvegarde un modèle et ses relations

```
$user->push();
```

Vous pouvez aussi lancer une mise à jour sur un ensemble de modèles :

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

Pour supprimer un modèle, appelez simplement la méthode `delete` sur une instance :

Suppression d'un modèle existant

```
$user = User::find(1);

$user->delete();
```

Suppression de modèles par leur clé

```
User::destroy(1);

User::destroy(array(1, 2, 3));

User::destroy(1, 2, 3);
```

Bien sûr, vous pouvez également supprimer un ensemble de modèles :

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

Si vous souhaitez simplement mettre à jour les timestamps d'un modèle, utilisez la méthode `touch` :

Mise à jour uniquement des timestamps d'un modèle

```
$user->touch();
```

Suppression douce

Lors de la suppression douce d'un modèle, il n'est en fait pas vraiment supprimé de votre base de données. A la place, un timestamp `deleted_at` est défini sur la ligne. Pour activer la suppression douce sur un modèle, ajoutez la propriété `softDelete` à ce dernier :

```
class User extends Eloquent {

    protected $softDelete = true;

}
```

Pour ajouter une colonne `deleted_at` à votre table, vous pouvez utiliser la méthode `softDeletes` depuis une migration:

```
$table->softDeletes();
```

Maintenant, lorsque vous appelez la méthode `delete` sur le modèle, la colonne `deleted_at` sera remplie avec la date et l'heure de suppression. Lorsque vous requêtez un modèle avec de la suppression douce, les modèles "supprimés" ne seront pas inclus dans le résultat. Pour forcer l'apparition des modèles réputés supprimés, utilisez la méthode `withTrashed` sur la requête :

Force l'affichage des lignes réputées supprimées

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

Si vous souhaitez recevoir **uniquement** les lignes supprimées, utilisez la méthode `onlyTrashed` :

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

Pour annuler cette suppression, utilisez la méthode `restore` :

```
$user->restore();
```

Vous pouvez également utiliser la méthode `restore` sur une requête :

```
User::withTrashed()->where('account_id', 1)->restore();
```

La méthode `restore` peut également être utilisée sur une relation :

```
$user->posts()->restore();
```

Si vous souhaitez réellement supprimer une ligne de la base de données, vous pouvez utiliser la méthode `forceDelete` :

```
$user->forceDelete();
```

La méthode `forceDelete` marche également sur les relations :

```
$user->posts()->forceDelete();
```

Pour déterminer si un modèle donné a été supprimé de manière douce, vous pouvez utiliser la méthode `trashed` :

```
if ($user->trashed()) {
    //
}
```

Timestamps

Par défaut, Eloquent maintiendra les colonnes `created_at` et `updated_at` de votre table automatiquement. Ajoutez simplement ces colonnes de type `timestamp` à votre table et Eloquent va automatiquement se charger du reste. Si vous ne souhaitez pas qu'Eloquent s'en occupe, ajoutez la propriété suivante au modèle :

Désactivation de l'auto-timestamps

```
class User extends Eloquent {

    protected $table = 'users';

    public $timestamps = false;

}
```

Si vous souhaitez personnaliser le format de vos timestamps, surchargez la méthode `getDateFormat` de votre modèle :

Création d'un format de timestamp personnalisé pour ce modèle

```
class User extends Eloquent {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

Cadres de requête

Les cadres vous permettent de réutiliser facilement des logiques de requêtes dans vos modèles. Pour définir un cadre, préfixez simplement une méthode du modèle avec `scope` :

Définition d'un cadre de requête

```
class User extends Eloquent {

    public function scopePopular($query)
    {
       return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }
}
```

Utilisation d'un cadre de requête

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Cadres dynamiques

Des fois, vous pouvez vouloir définir un cadre qui accepte des paramètres. Ajoutez juste vos paramètres dans la fonction de cadre :

```
class User extends Eloquent {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

Passez ensuite le paramètre dans l'appel du cadre :

```
$users = User::ofType('member')->get();
```

Relations

Bien sûr, vos tables sont probablement liées les unes aux autres. Par exemple, un post de blog a plusieurs commentaires, ou une commande est liée à l'utilisateur qui l'a passée. Eloquent rend la gestion et le travail avec ces relations simples. Laravel supporte quatre types de relation :

- Un vers un (1:1)
- Un vers plusieurs (1:n)
- Plusieurs vers plusieurs (n:n)
- Plusieurs via
- Relations polymorphiques
- Relations polymorphique plusieurs vers plusieurs

Un vers un (1:1)

Une relation un-vers-un est une relation très basique. Par exemple, un modèle `User` peut avoir un modèle `Phone` . Nous définissons la relation de la manière suivante avec Eloquent :

Définition d'une relation un vers un

```
class User extends Eloquent {

    public function phone()
```

```
    {
        return $this->hasOne('Phone');
    }

}
```

Le premier argument passé à la méthode `hasOne` est le nom du modèle lié. Une fois que la relation est définie, nous pouvons la récupérer en utilisant les propriétés dynamiques d'Eloquent :

```
$phone = User::find(1)->phone;
```

La requête SQL exécutée pour cette requête sera la suivante :

```
select * from users where id = 1

select * from phones where user_id = 1
```

Notez qu'Eloquent devine la clé étrangère en se basant sur le nom du modèle. Dans ce cas, le modèle `Phone` doit avoir une colonne `user_id` en tant que clé étrangère. Vous pouvez surcharger cette convention en passant un second argument à la méthode `hasOne` . De plus, vous pouvez ajouter un troisième argument à la méthode pour spécifier quelle colonne locale doit être utilisée pour l'association :

```
return $this->hasOne('Phone', 'foreign_key');

return $this->hasOne('Phone', 'foreign_key', 'local_key'););
```

Définition de la relation inverse

Pour définir la relation inverse sur le modèle `Phone` , nous utilisons la méthode `belongsTo` :

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User');
    }

}
```

Dans l'exemple ci dessus, Eloquent recherchera une colonne `user_id` sur la table `phones` . Si vous souhaitez définir un autre nom pour votre clé étrangère, vous pouvez le passer en tant que second argument de la méthode `belongsTo` :

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User', 'local_key');
    }

}
```

Additionnellement, vous pouvez passer un troisième paramètre qui spécifie le nom de la colonne associée dans la table parent :

class Phone extends Eloquent {

```
  public function user()
  {
    return $this->belongsTo('User', 'local_key', 'parent_key');
  }


}
```

Un vers plusieurs (1:n)

Un exemple de relation un-vers-plusieurs est un post de blog qui a plusieurs commentaires. Nous réalisons cette relation comme cela :

```
class Post extends Eloquent {

    public function comments()
    {
        return $this->hasMany('Comment');
    }

}
```

Nous pouvons accéder aux commentaires du post via la propriété dynamique :

```
$comments = Post::find(1)->comments;
```

Si vous avez besoin d'ajouter des contraintes supplémentaires à la récupération de 'comments', appelez la méthode `comments` et continuez à chainer les conditions :

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

Une fois encore, vous pouvez surcharger le nom de la clé étrangère en passant en tant que second argument son nom à la méthode `hasMany` . Et, comme pour la relation `hasOne` , la colonne locale peut également être spécifiée :

```
return $this->hasMany('Comment', 'foreign_key');

return $this->hasMany('Comment', 'foreign_key', 'local_key');
```

Pour définir la relation inverse, sur le modèle `Comment` , nous utilisons la méthode `belongsTo` :

Définition de la relation inverse

```
class Comment extends Eloquent {

    public function post()
    {
        return $this->belongsTo('Post');
    }

}
```

Plusieurs vers plusieurs (n:n)

Les relations plusieurs-vers-plusieurs sont un type un peu plus compliqué. Par exemple un utilisateur peut avoir plusieurs rôles, et un rôle peut être assigné à plusieurs utilisateurs. Plusieurs utilisateurs peuvent avoir un rôle "Admin" par exemple. Pour établir cette relation, nous avons besoin de trois tables : `users` , `roles` , et `role_user` . La table `role_user` est dérivée de l'ordre alphabétique des modèles liés, et doit contenir les colonnes `user_id` et `role_id` .

Nous pouvons définir une relation de type plusieurs-vers-plusieurs en utilisant la méthode `belongsToMany` :

```
class User extends Eloquent {

    public function roles()
    {
        return $this->belongsToMany('Role');
    }

}
```

Maintenant nous pouvons récupérer nos rôles via le modèle `User` :

```
$roles = User::find(1)->roles;
```

Si vous souhaitez utiliser un nom non conventionnel pour votre table pivot, passez le second argument de la méthode `belongsToMany` :

```
return $this->belongsToMany('Role', 'user_roles');
```

Vous pouvez également surcharger les clés associées :

```
return $this->belongsToMany('Role', 'user_roles', 'user_id', 'foo_id');
```

Bien sûr, vous pouvez aussi avoir besoin de définir de la relation dans le modèle `Role` :

```
class Role extends Eloquent {

    public function users()
    {
        return $this->belongsToMany('User');
    }

}
```

Plusieurs via

La relation "plusieurs via" fournit un raccourci pratique pour accéder à une relation distante via une relation intermédiaire. Par exemple, un modèle `Country` peut avoir plusieurs `Posts` via un modèle `Users` . Les tables pour cette relation ressemble à cela :

```
countries
  id - integer
  name - string

users
  id - integer
  country_id - integer
  name - string

posts
  id - integer
  user_id - integer
  title - string
```

Même si la table `posts` le contient pas une colonne `country_id` , la relation `hasManyThrough` ne donne la possibilité d'accéder aux posts d'un pays via `$country->posts` . Définissons la relation :

```
class Country extends Eloquent {

    public function posts()
    {
        return $this->hasManyThrough('Post', 'User');
    }

}
```

Si vous souhaitez définir manuellement les clés de la relation, vous pouvez le faire grâce au troisième et au quatrième argument de la méthode :

```
class Country extends Eloquent {

    public function posts()
    {
        return $this->hasManyThrough('Post', 'User', 'country_id', 'user_id');
    }

}
```

Relations polymorphiques

Les relations polymorphiques permettent à un modèle d'appartenir à plus d'un autre modèle, en une simple association. Par exemple, vous pourriez avoir un modèle Photo qui appartient au modèle Staff ainsi qu'au modèle Commande. Nous définirons cette relation de la manière suivante :

```
class Photo extends Eloquent {

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

}

class Staff extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}

class Commande extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

Maintenant, nous pouvons récupérer les photos soit de notre staff, soit d'une commande :

Récupération d'une relation polymorphique

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo) {
    //
}
```

Récupération du propriétaire de la Photo

Cependant, la vraie magie de la polymorphie apparaît lorsque vous accédez au staff ou à la commande depuis le modèle `Photo` :

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

La relation `imageable` du modèle `Photo` retournera soit une instance de `Staff` ou de `Commande`, selon le modèle propriétaire de la photo.

Pour vous aider à comprendre comment cela marche, jetons un oeil à la structure de la base de données pour une relation polymorphique :

Structure de la base de données pour une relation polymorphique

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
```

```
    imageable_id - integer
    imageable_type - string
```

Les champs clés à remarquer ici sont `imageable_id` et `imageable_type` de la table `photos`. L'ID contiendra la valeur de l'ID d'une ligne de staff ou de commande ici par exemple, tandis que le type contiendra le nom de la classe du modèle propriétaire. C'est ce qui permet à l'ORM de déterminer quel type de propriétaire doit être retourné lors de l'accès à la relation `imageable`.

Relations polymorphique plusieurs vers plusieurs

En plus des relations polymorphiques traditionnelles, vous pouvez créer des relations polymorphiques plusieurs vers plusieurs. Par exemple, un post de blog et une vidéo peuvent partager une relation avec des `Tag`. Premièrement, regardons la structure des tables:

Structure de tables d'une relations polymorphic plusieurs vers plusieurs

```
posts
        id - integer
        name - string

videos
        id - integer
        name - string

tags
        id - integer
        name - string

taggables
        tag_id - integer
        taggable_id - integer
        taggable_type - string
```

Ensuite, nous sommes prêt à mettre en place la relations dans le modèle. Les modèles `Post` et `Video` auront tous les deux une relation `morphToMany` via une méthode `tags` :

```
class Post extends Eloquent {

        public function tags()
        {
                return $this->morphToMany('Tag', 'taggable');
        }

}
```

Le modèle `Tag` doit définir une méthode pour chacune de ses relations :

```
class Tag extends Eloquent {

        public function posts()
        {
                return $this->morphedByMany('Post', 'taggable');
        }

        public function videos()
        {
                return $this->morphedByMany('Video', 'taggable');
        }

}
```

Requêtes sur les relations

Lorsque vous accédez aux lignes d'un modèle, vous pourriez vouloir limiter vos résultats en se basant sur l'existence d'une relation. Par exemple, pour récupérer les billets d'un blog qui ont au moins un commentaire. Pour ce faire, vous pouvez utiliser la méthode `has` :

Requête d'une relation lors de la sélection

```
$posts = Post::has('comments')->get();
```

Vous pouvez également spécifier un opérateur et un nombre :

```
$posts = Post::has('comments', '>=', 3)->get();
```

Si vous voulez encore plus de puissance, vous pouvez utiliser `whereHas` et `orWhereHas` pour placer une condition "where" sur une requête `has` :

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');
})->get();
```

Propriétés dynamiques

Eloquent vous autorise à accéder à vos relations par des propriétés dynamiques. Eloquent va automatiquement charger la relation pour vous, et est assez malin pour savoir quand appeler la méthode `get` (pour les relations one-to-many) ou `first` (pour les relations one-to-one). La relation sera alors accessible par une propriété dynamique qui porte le même nom que la relation. Par exemple, avec le modèle `$phone` :

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User');
    }

}

$phone = Phone::find(1);
```

Plutôt que d'afficher l'adresse e-mail de l'utilisateur ainsi :

```
echo $phone->user()->first()->email;
```

L'appel peut se faire de cette manière :

```
echo $phone->user->email;
```

> **Note:** les relations retournant plusieurs résultats retourneront une instance de la classe `Illuminate\Database\Eloquent\Collection` .

Chargements liés

Les chargements liés (eager loading) existent pour éviter le problème des requêtes N + 1. Par exemple, disons qu'un modèle `Book` est relié à un modèle `Author` . La relation est définie de la manière suivante :

```
class Book extends Eloquent {

    public function author()
    {
        return $this->belongsTo('Author');
    }

}
```

Maintenant considérez le code suivant :

```
foreach (Book::all() as $book) {
    echo $book->author->name;
}
```

La boucle exécutera une requête pour récupérer tous les livres de la table, ensuite une autre requête sur chaque livre pour récupérer l'auteur. Donc, si nous avons 25 livres, nous aurons 26 requêtes.

Heureusement, nous pouvons utiliser les chargements liés pour réduire drastiquement le nombre de requêtes. Les relations qui doivent être chargées doivent être précisées avec la méthode `with` :

```
foreach (Book::with('author')->get() as $book) {
    echo $book->author->name;
}
```

Pour la boucle ci-dessus, les requêtes suivantes sont exécutées :

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Une utilisation sage des chargements liés peut augmenter drastiquement les performances de votre application.

Bien sûr, vous pouvez faire des chargements liés sur plusieurs relations en une fois :

```
$books = Book::with('author', 'publisher')->get();
```

Vous pouvez même faire du chargement lié de manière imbriquée :

```
$books = Book::with('author.contacts')->get();
```

Dans l'exemple ci-dessus, la relation `author` sera chargée de manière liée, et les contacts de l'auteur seront chargés également.

Contraintes sur les chargements liés

Si vous avez besoin d'ajouter des contraintes sur un chargement lié, vous pouvez le faire de la manière suivante :

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');
}))->get();
```

Dans cet exemple, nous chargeons les posts de l'utilisateur, mais seulement si le post contient le mot "first".

Chargements liés différés

Il est également possible de faire du chargement lié directement sur une collection de modèles existants. Cela peut s'avérer utile si vous devez décider dynamiquement de charger les modèles liés ou non, ou en combinaison avec du cache.

```
$books = Book::all();

$books->load('author', 'publisher');
```

Insertion de modèles liés

Vous aurez souvent besoin d'insérer des nouveaux modèles liés. Par exemple, pour insérer un commentaire lié à un post de blog, plutôt que de définir manuellement la clé étrangère `post_id` sur le modèle, vous pouvez insérer un nouveau commentaire directement depuis son modèle parent `Post` :

Attachement à un modèle lié

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

Dans cet exemple, le champ `post_id` sera automatiquement rempli dans le commentaire inséré.

Associations de modèles (Belongs To)

Lors de la mise à jour d'une relation `belongsTo`, vous pouvez utiliser la méthode `associate`. Cette méthode renseigne la clé étrangère dans le modèle enfant :

```
$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

Insertion de modèles liés, plusieurs vers plusieurs

Vous devrez également insérer des modèles liés par une relation `plusieurs vers plusieurs`. Continuons d'utiliser nos modèles d'exemples `User` et `Role`. Nous pouvons facilement attacher des nouveaux rôles à un utilisateur avec la méthode :

Attache des modèles liés par une relation plusieurs vers plusieurs

```
$user = User::find(1);

$user->roles()->attach(1);
```

Vous pouvez également passer un tableau d'attributs qui doivent être stockés dans la table pivot pour la relation :

```
$user->roles()->attach(1, array('expires' => $expires));
```

Naturellement, l'opposé de `attach` est `detach` :

```
$user->roles()->detach(1);
```

Vous pouvez également utiliser la méthode `sync` pour attacher des modèles liés. La méthode `sync` accepte un tableau d'IDs à placer dans la table pivot. Une fois cette opération terminée, seuls les IDs dans le tableau seront dans la table pivot pour le modèle :

Utilisation de la méthode Sync pour attacher des modèles liés

```
$user->roles()->sync(array(1, 2, 3));
```

Vous pouvez également créer un nouveau modèle lié et l'attacher en une simple ligne. Pour cette opération, utilisez la méthode `save` :

```
$role = new Role(array('name' => 'Editor'));

User::find(1)->roles()->save($role);
```

Dans cet exemple, le nouveau modèle `Role` sera sauvegardé et attaché au modèle `User`. Vous pourriez également avoir besoin de passer un tableau d'attributs pour le sauvegarder dans la table de jointure :

```
User::find(1)->roles()->save($role, array('expires' => $expires));
```

Mise à jour du Timestamps des parents

Quand un modèle appartient à (`belongsTo`) un autre modèle, comme un `Comment` appartient à un `Post` , il est souvent utile de mettre à jour les timestamps du parent lorsque le modèle enfant est mis à jour. Par exemple, lorsqu'un commentaire est modifié, nous pourrions mettre à jour le `Post` qui le contient. Eloquent rend cela facile. Ajoutez simplement la propriété `touches` qui contient le nom des relations dans le modèle enfant :

```
class Comment extends Eloquent {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('Post');
    }

}
```

Maintenant, quand vous mettrez à jour un `Comment` , le `Post` parent aura sa colonne `updated_at` mise à jour :

```
$comment = Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

Travail sur les tables pivots

Comme vous l'avez déjà appris, le travail avec les relations plusieurs-vers-plusieurs nécessite la présence d'une table intermédiaire. Eloquent fournit des moyens très utiles d'interagir avec cette table. Par exemple, disons que nous avons un objet `User` qui a plusieurs objets `Role` . Après avoir accédé à la relation, vous pouvez accéder à la table `pivot` du modèle :

```
$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notez que chaque modèle `Role` que nous récupérons aura automatiquement l'attribut `pivot` assigné. Cet attribut contient un modèle qui représente la table intermédiaire, et peut être utilisé comme n'importe quel autre modèle Eloquent.

Par défaut, seules les clés seront présentes dans l'objet `pivot` . Si votre table pivot contient des attributs en plus, vous devez les spécifier lors de la définition de la relation :

```
return $this->belongsToMany('Role')->withPivot('foo', 'bar');
```

Maintenant, les attributs `foo` et `bar` seront accessibles par l'objet `pivot` pour le modèle `Role` .

Si vous souhaitez que votre table pivot ait les timestamps `created_at` et `updated_at` automatiquement maintenus, utilisez la méthode `withTimestamps` sur la définition de la relation :

```
return $this->belongsToMany('Role')->withTimestamps();
```

Pour supprimer toutes les lignes de la table pivot pour un modèle, vous pouvez utiliser la méthode `detach` :

Suppression des lignes de la table pivot

```
User::find(1)->roles()->detach();
```

Notez que cette opération ne supprimera pas les enregistrements de la table `roles` , mais seulement de la table pivot.

Définition d'un modèle de pivot personnalisé

Laravel vous permet de définir votre propre modèle de pivot. Pour ce faire, créez votre propose modèle de "Base" qui étend `Eloquent`. Dans vos autres modèles Eloquent, héritez de cette "Base" plutôt que d' `Eloquent`. Dans votre modèle de "Base", Ajoutez la fonction suivante qui retourne une instance de de votre modèle de pivot personnalisé :

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
}
```

Collections

Toutes les requêtes qui renvoient plusieurs résultats par Eloquent, via la méthode `get` ou par une relation, retournent un objet `Collection`. Cet objet implémente l'interface PHP `IteratorAggregate`, donc nous pouvons itérer dessus comme pour un tableau. Cependant, cet objet a également une variété de méthodes utiles pour travailler avec une liste de résultats.

Par exemple, nous pouvons déterminer si une liste de résultats contient une clé primaire donnée en utilisant la méthode `contains` :

Vérifie si une collection contient une clé

```
$roles = User::find(1)->roles;

if ($roles->contains(2)) {
    //
}
```

Les Collections peuvent être converties en tableau ou en JSON :

```
$roles = User::find(1)->roles->toArray();

$roles = User::find(1)->roles->toJson();
```

Si une collection est castée en une chaîne, alors sa représentation JSON sera retournée :

```
$roles = (string) User::find(1)->roles;
```

Les collections Eloquent contiennent également quelques méthodes utiles pour boucler et filtrer sur les objets qu'elle contient :

Bouclage de collections

```
$roles = $user->roles->each(function($role)
{

});
```

Filtrage de collections

Lors du filtrage de collections, le retour fourni sera utilisé comme retour pour array_filter.

```
$users = $users->filter(function($user)
{
    if ($user->isAdmin()) {
        return $user;
    }
});
```

> **Note:** lors du filtrage d'une collection et la conversion en JSON, essayez d'appeler les `valeurs` de la première fonction pour remettre à zéro les clés du tableau.

Applique une fonction sur chaque object d'une collection

```
$roles = User::find(1)->roles;

$roles->each(function($role)
{

});
```

Tri une collection par une valeur

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});
```

Tri une collection par une valeur

```
$roles = $roles->sortBy('created_at');
```

Parfois, vous pourriez vouloir retourner une collection personnalisée avec vos propres méthodes ajoutées. Vous devez spécifier cela dans votre modèle Eloquent en surchargeant la méthode `newCollection` :

Retourne un type de collection personnalisé

```
class User extends Eloquent {

    public function newCollection(array $models = array())
    {
        return new CustomCollection($models);
    }

}
```

Les accesseurs et mutateurs

Eloquent fournit une manière efficace de transformer vos attributs de modèle lorsque vous les récupérez ou les définissez. Définissez simplement une méthode `getFooAttribute` sur votre modèle pour créer un accesseur. Gardez à l'esprit que les méthodes doivent être en camelCase, même si les colonnes de votre base sont en snake_case :

Définition d'un accesseur

```
class User extends Eloquent {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

Dans l'exemple ci-dessus, la colonne `first_name` a un accesseur. Notez que la valeur est passée à l'accesseur.

Les mutateurs sont déclarés dans le même esprit :

Définition d'un mutateur

```
class User extends Eloquent {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

Mutateurs de date

Par défaut, Eloquent va convertir les colonnes `created_at`, `updated_at` et `deleted_at` en instance de Carbon, qui fournit un lôt de méthodes utiles, et hérite de la classe PHP `DateTime`.

Vous pouvez personnaliser quels champs seront automatiquement mutés, voir même désactiver cette mutation, en surchargeant la méthode du modèle :

```
public function getDates()
{
    return array('created_at');
}
```

Lorsqu'une colonne est considérée comme une date, vous pouvez définir sa valeur comme étant un timestamp UNIX, une chaîne de date (`Y-m-d`), une chaîne de type date et heure (`Y-m-d H:i:s`), ou bien sûr une instance de `DateTime` / `Carbon`.

Pour désactiver totalement la mutation des dates, retournez simplement un tableau vide depuis la méthode `getDates` :

```
public function getDates()
{
    return array();
}
```

Evénements de modèle

Les modèles Eloquent lancent plusieurs événements, vous permettant d'interagir avec le modèle durant son cycle de vie en utilisant les méthodes : `creating` (avant la création), `created` (une fois créé), `updating` (avant la mise à jour), `updated` (une fois mis à jour), `saving` (avant l'enregistrement), `saved` (une fois enregistré), `deleting` (avant la suppression), `deleted` (une fois supprimé), `restoring` (avant la restauration), `restored` (une fois restauré).

Chaque fois qu'un nouvel item est sauvegardé pour la première fois, les événements `creating` et `created` sont lancés. Si un item n'est pas nouveau et que la méthode `save` est appelée, les événements `updating` / `updated` sont lancés. Dans les deux cas, les événements `saving` / `saved` sont lancés.

Si `false` est retourné par la méthode `creating`, `updating`, ou `saving`, alors l'action est annulée :

Annulation de la création d'un modèle

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

Les modèles Eloquent contiennent également une méthode static `boot`, qui peut être l'endroit idéal pour s'abonner aux événements.

Mise en place de la méthode boot d'un modèle

```
class User extends Eloquent {

    public static function boot()
    {
        parent::boot();

        // Setup event bindings...
  }

}
```

Observateurs de modèle

Pour consolider le gestion des événements d'un modèle, vous pouvez enregistrer une observateur de modèle. Une classe d'observation peut avoir des méthodes qui correspondent à plusieurs événements d'un modèle. Par exemple, les méthodes `creating`, `updating`, `saving` peuvent être sur un observateur, en plus de n'importe quel autre nom d'événement de modèle.

Donc, par exemple, un observateur de modèle peut ressembler à cela :

```
class UserObserver {

    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
        //
    }

}
```

Vous pouvez enregistrer une instance d'un observateur en utilisant la méthode `observe` :

```
User::observe(new UserObserver);
```

Conversion en tableau / JSON

Quand vous construisez des APIs en JSON, vous devez souvent convertir vos modèles et vos relations en tableau ou en JSON. Eloquent inclut des méthodes pour le faire. Pour convertir un modèle et ses relations en tableau, vous pouvez utiliser la méthode `toArray` :

Convertion d'un modèle en tableau

```
$user = User::with('roles')->first();

return $user->toArray();
```

Notez que l'intégralité des collections de modèles peuvent être converties en tableau :

```
return User::all()->toArray();
```

Pour convertir un modèle en JSON, vous pouvez utiliser la méthode `toJson` :

Conversion d'un modèle en JSON

```
return User::find(1)->toJson();
```

Notez que quand un modèle ou une collection est casté en string, ils sont convertis en JSON, ce qui signifie que vous pouvez retourner des objets Eloquent directement depuis vos routes/actions !

Retourne un modèle depuis une route

```
Route::get('users', function()
{
    return User::all();
});
```

Parfois vous pourriez souhaiter que certains attributs ne soient pas inclus dans la forme tableau ou JSON de vos modèles, tels que les mots de passes. Pour ce faire, ajoutez la propriété `hidden` à la définition de votre modèle :

Cache un attribut des formats tableaux ou JSON

```
class User extends Eloquent {

    protected $hidden = array('password');

}
```

> **Note:** Quand vous cachez des relations, utilisez le nom de la **méthode** de la relation, pas le nom d'accession dynamique.

Alternativement, vous pouvez utiliser la propriété `visible` pour définir une liste blanche :

```
protected $visible = array('first_name', 'last_name');
```

Occasionnellement, vous pouvez avoir besoin d'ajouter un tableau d'attributs qui ne correspondent pas à un colonne dans votre base de données. Pour ce faire, vous devez simplement définir un accesseur pour la valeur :

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

Une fois que vous avez créé l'accesseur, ajoutez la valeur de la propriété `appends` au modèle :

```
protected $appends = array('is_admin');
```

Une fois que l'attribut a été ajouté à la liste `appends`, il peut être inclus à la fois sous la modèle tableau ou JSON du modèle.