Base de données: Générateur de requêtes

#introduction

Le constructeur de requêtes de base de données de Laravel fournit une interface pratique et fluide pour la création et l'exécution de requêtes de base de données. Il peut être utilisé pour effectuer la plupart des opérations de base de données dans votre application et fonctionne sur tous les systèmes de base de données pris en charge.

Le générateur de requêtes Laravel utilise la liaison de paramètres PDO pour protéger votre application contre les attaques par injection SQL. Il n'est pas nécessaire de nettoyer les chaînes passées en tant que liaisons.

#Récupération des résultats

Récupérer toutes les lignes d'une table

Vous pouvez utiliser la `table` méthode sur la `DB` façade pour lancer une requête. La `table` méthode retourne une instance de générateur de requête fluide pour la table donnée, vous permettant ainsi d'enchaîner plus de contraintes sur la requête, puis d'obtenir les résultats à l'aide de la `get` méthode:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

La `get` méthode retourne un contenant les résultats, chaque résultat étant une instance de l' objet PHP . Vous pouvez accéder à la valeur de chaque colonne en accédant à la colonne en tant que propriété de l'objet: `Illuminate\Support\Collection` `stdClass`

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

```
}
```

Récupérer une seule ligne / colonne d'une table

Si vous devez simplement extraire une seule ligne de la table de la base de données, vous pouvez utiliser la `first` méthode. Cette méthode renverra un seul `stdClass` objet:

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

Si vous n'avez même pas besoin d'une ligne entière, vous pouvez extraire une valeur unique d'un enregistrement à l'aide de la `value` méthode. Cette méthode renverra directement la valeur de la colonne:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Récupérer une liste de valeurs de colonnes

Si vous souhaitez récupérer une collection contenant les valeurs d'une seule colonne, vous pouvez utiliser la `pluck` méthode. Dans cet exemple, nous allons récupérer une collection de titres de rôles:

```
$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

Vous pouvez également spécifier une colonne de clé personnalisée pour la collection renvoyée:

```
$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

Résultats tronqués

Si vous devez travailler avec des milliers d'enregistrements de base de données, envisagez d'utiliser la `chunk` méthode. Cette méthode récupère une petite partie des résultats à la fois et alimente chaque

partie en `Closure` traitement. Cette méthode est très utile pour écrire des commandes Artisan traitant des milliers d'enregistrements. Par exemple, travaillons avec la `users` table entière par tranches de 100 enregistrements à la fois:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

Vous pouvez empêcher le traitement d'autres morceaux en retournant `false` du `Closure` :

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Process the records...

    return false;
});
```

Si vous mettez à jour des enregistrements de base de données tout en regroupant les résultats, ceux-ci pourraient être modifiés de manière inattendue. Ainsi, lors de la mise à jour d'enregistrements tout en segmentant, il est toujours préférable d'utiliser la `chunkById` méthode à la place. Cette méthode paginera automatiquement les résultats en fonction de la clé primaire de l'enregistrement:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

> Lors de la mise à jour ou de la suppression d'enregistrements dans le rappel de bloc, toute modification apportée à la clé primaire ou à une clé étrangère peut affecter la requête de bloc. Cela pourrait entraîner que des enregistrements ne soient pas inclus dans les résultats découpés.

Agrégats

Le générateur de requêtes fournit également une variété de méthodes globales , telles que `count` , `max` , `min` , `avg` et `sum` . Vous pouvez appeler n'importe laquelle de ces méthodes après avoir construit votre requête:

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Bien entendu, vous pouvez combiner ces méthodes avec d'autres clauses:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Déterminer si des enregistrements existent

Au lieu d'utiliser la `count` méthode pour déterminer s'il existe des enregistrements correspondant aux contraintes de votre requête, vous pouvez utiliser les méthodes `exists` et `doesntExist` :

```
return DB::table('orders')->where('finalized', 1)->exists();

return DB::table('orders')->where('finalized', 1)->doesntExist();
```

Sélectionne

Spécification d'une clause de sélection

Bien entendu, il se peut que vous ne souhaitiez pas toujours sélectionner toutes les colonnes d'une table de base de données. En utilisant la `select` méthode, vous pouvez spécifier une `select` clause personnalisée pour la requête:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

La `distinct` méthode vous permet de forcer la requête à renvoyer des résultats distincts:

```
$users = DB::table('users')->distinct()->get();
```

Si vous avez déjà une instance de générateur de requête et que vous souhaitez ajouter une colonne à sa clause select existante, vous pouvez utiliser la `addSelect` méthode suivante:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

#Expressions brutes

Parfois, vous devrez peut-être utiliser une expression brute dans une requête. Pour créer une expression brute, vous pouvez utiliser la méthode suivante: `DB::raw`

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

> Les instructions brutes seront injectées dans la requête sous forme de chaînes. Vous devez donc faire très attention de ne pas créer de vulnérabilités d'injection SQL.

Méthodes brutes

Au lieu d'utiliser , vous pouvez également utiliser les méthodes suivantes pour insérer une expression brute dans différentes parties de votre requête. `DB::raw`

`selectRaw`

La `selectRaw` méthode peut être utilisée à la place de . Cette méthode accepte un tableau optionnel de liaisons comme second argument: `select(DB::raw(...))`

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

`whereRaw / orWhereRaw`

Les méthodes `whereRaw` et `orWhereRaw` peuvent être utilisées pour injecter une `where` clause brute dans votre requête. Ces méthodes acceptent un tableau facultatif de liaisons comme deuxième argument:

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
                ->get();
```

`havingRaw / orHavingRaw`

Les méthodes `havingRaw` et `orHavingRaw` peuvent être utilisées pour définir une chaîne brute en tant que valeur de la `having` clause. Ces méthodes acceptent un tableau facultatif de liaisons comme deuxième argument:

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

`orderByRaw`

La `orderByRaw` méthode peut être utilisée pour définir une chaîne brute comme valeur de la `order by` clause:

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

Jointures

Clause de jointure interne

Le constructeur de requêtes peut également être utilisé pour écrire des instructions de jointure. Pour effectuer une "jointure interne" de base, vous pouvez utiliser la `join` méthode sur une instance du générateur de requêtes. Le premier argument transmis à la `join` méthode est le nom de la table à laquelle vous devez vous connecter, tandis que les autres arguments spécifient les contraintes de colonne pour la jointure. Bien sûr, comme vous pouvez le constater, vous pouvez joindre plusieurs tables en une seule requête:

```
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Clause jointe à gauche

Si vous souhaitez effectuer une "jointure gauche" au lieu d'une "jointure interne", utilisez la `leftJoin` méthode. La `leftJoin` méthode a la même signature que la `join` méthode:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Clause de jonction

Pour effectuer une "jointure croisée", utilisez la `crossJoin` méthode portant le nom de la table à laquelle vous souhaitez participer. Les jointures croisées génèrent un produit cartésien entre la première table et la table jointe:

```
$users = DB::table('sizes')
            ->crossJoin('colours')
            ->get();
```

Clauses de jointure avancées

Vous pouvez également spécifier des clauses de jointure plus avancées. Pour commencer, passez un `Closure` second argument à la `join` méthode. Le `Closure` recevra un `JoinClause` objet qui vous permettra de spécifier des contraintes sur la `join` clause:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

Si vous souhaitez utiliser une clause de style "where" sur vos jointures, vous pouvez utiliser les méthodes `where` and `orWhere` sur une jointure. Au lieu de comparer deux colonnes, ces méthodes comparent la colonne à une valeur:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Jointures de sous-requêtes

Vous pouvez utiliser les `joinSub`, `leftJoinSub` et les `rightJoinSub` méthodes pour se joindre à une requête à un sous-requête. Chacune de ces méthodes reçoit trois arguments: la sous-requête, son alias de table et une fermeture définissant les colonnes associées:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
                   ->where('is_published', true)
                   ->groupBy('user_id');

$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function ($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

#Les syndicats

Le constructeur de requêtes fournit également un moyen rapide de "réunir" deux requêtes. Par exemple, vous pouvez créer une requête initiale et utiliser la `union` méthode pour la joindre à une seconde requête:

```
$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

> La `unionAll` méthode est également disponible et a la même signature de méthode que `union`.

#Où les clauses

Clauses simples où

Vous pouvez utiliser la `where` méthode sur une instance de générateur de requête pour ajouter des `where` clauses à la requête. L'appel le plus élémentaire à `where` nécessite trois arguments. Le premier argument est le nom de la colonne. Le deuxième argument est un opérateur, qui peut être n'importe quel opérateur pris en charge par la base de données. Enfin, le troisième argument est la valeur à évaluer par rapport à la colonne.

Par exemple, voici une requête qui vérifie que la valeur de la colonne "votes" est égale à 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

Pour plus de commodité, si vous souhaitez vérifier qu'une colonne est égale à une valeur donnée, vous pouvez passer directement la valeur en tant que second argument de la `where` méthode:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Bien sûr, vous pouvez utiliser plusieurs autres opérateurs lors de la rédaction d'une `where` clause:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

Vous pouvez également transmettre un tableau de conditions à la `where` fonction:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
```

```
    ['subscribed', '<>', '1'],
])->get();
```

Ou des déclarations

Vous pouvez chaîner les contraintes et ajouter des `or` clauses à la requête. La `orWhere` méthode accepte les mêmes arguments que la `where` méthode:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

Clauses additionnelles

oùEntre

La `whereBetween` méthode vérifie que la valeur d'une colonne est comprise entre deux valeurs:

```
$users = DB::table('users')
                    ->whereBetween('votes', [1, 100])->get();
```

oùNe pas Entre

La `whereNotBetween` méthode vérifie que la valeur d'une colonne se situe en dehors de deux valeurs:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

oùIn / oùNonIn

La `whereIn` méthode vérifie que la valeur d'une colonne donnée est contenue dans le tableau donné:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

La `whereNotIn` méthode vérifie que la valeur de la colonne donnée n'est **pas** contenue dans le tableau donné:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

oùNull / WhereNotNull

La `whereNull` méthode vérifie que la valeur de la colonne donnée est `NULL` :

```
$users = DB::table('users')
                    ->whereNull('updated_at')
                    ->get();
```

La `whereNotNull` méthode vérifie que la valeur de la colonne n'est pas `NULL` :

```
$users = DB::table('users')
                    ->whereNotNull('updated_at')
                    ->get();
```

oùDate / whereMoish / whereDay / whereYear / whereTime

La `whereDate` méthode peut être utilisée pour comparer la valeur d'une colonne à une date:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

La `whereMonth` méthode peut être utilisée pour comparer la valeur d'une colonne à un mois spécifique de l'année:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

La `whereDay` méthode peut être utilisée pour comparer la valeur d'une colonne à un jour spécifique d'un mois:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

La `whereYear` méthode peut être utilisée pour comparer la valeur d'une colonne à une année spécifique:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

La `whereTime` méthode peut être utilisée pour comparer la valeur d'une colonne à une heure spécifique:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

oùColonne

La `whereColumn` méthode peut être utilisée pour vérifier que deux colonnes sont égales:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
                ->get();
```

Vous pouvez également passer un opérateur de comparaison à la méthode:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

La `whereColumn` méthode peut également recevoir un tableau de plusieurs conditions. Ces conditions seront jointes à l'aide de l' `and` opérateur:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at']
                ])->get();
```

Regroupement de paramètres

Parfois, vous devrez peut-être créer des clauses where plus avancées telles que les clauses "where exist" ou des groupes de paramètres imbriqués. Le constructeur de requêtes Laravel peut également les

gérer. Pour commencer, examinons un exemple de regroupement de contraintes entre parenthèses:

```
DB::table('users')
            ->where('name', '=', 'John')
            ->where(function ($query) {
                $query->where('votes', '>', 100)
                      ->orWhere('title', '=', 'Admin');
            })
            ->get();
```

Comme vous pouvez le constater, en passant un `Closure` dans la `where` méthode, le constructeur de requêtes demande au groupe de créer un groupe de contraintes. Le `Closure` recevra une instance de générateur de requête que vous pourrez utiliser pour définir les contraintes devant figurer dans le groupe de parenthèses. L'exemple ci-dessus produira le code SQL suivant:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

> Vous devez toujours regrouper les `orWhere` appels afin d'éviter tout comportement inattendu lorsque des étendues globales sont appliquées.

Clauses Where Exists

La `whereExists` méthode vous permet d'écrire `where exists` des clauses SQL. La `whereExists` méthode accepte un `Closure` argument, qui recevra une instance du générateur de requête vous permettant de définir la requête à placer dans la clause "existe":

```
DB::table('users')
            ->whereExists(function ($query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

La requête ci-dessus produira le code SQL suivant:

```
select * from users
where exists (
```

```
    select 1 from orders where orders.user_id = users.id
)
```

JSON Où Clauses

Laravel prend également en charge l'interrogation des types de colonnes JSON sur des bases de données prenant en charge les types de colonnes JSON. Actuellement, cela inclut MySQL 5.7, PostgreSQL, SQL Server 2016 et SQLite 3.9.0 (avec l' extension JSON1). Pour interroger une colonne JSON, utilisez l' opérateur: `->`

```
$users = DB::table('users')
                ->where('options->language', 'en')
                ->get();

$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

Vous pouvez utiliser `whereJsonContains` pour interroger des tableaux JSON (non pris en charge sur SQLite):

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

MySQL et PostgreSQL prennent `whereJsonContains` en charge plusieurs valeurs:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

Vous pouvez utiliser `whereJsonLength` pour interroger des tableaux JSON par leur longueur:

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
```

```
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

#Commande, regroupement, limite et décalage

commandé par

La `orderBy` méthode vous permet de trier le résultat de la requête en fonction d'une colonne donnée. Le premier argument de la `orderBy` méthode doit être la colonne que vous souhaitez trier par, tandis que le second argument contrôle le sens du tri et peut être soit `asc` ou `desc` :

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

le plus récent / le plus ancien

Les méthodes `latest` et `oldest` vous permettent de classer facilement les résultats par date. Par défaut, le résultat sera ordonné par la `created_at` colonne. Vous pouvez également indiquer le nom de la colonne que vous souhaitez trier:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

inRandomOrder

La `inRandomOrder` méthode peut être utilisée pour trier les résultats de la requête de manière aléatoire. Par exemple, vous pouvez utiliser cette méthode pour récupérer un utilisateur aléatoire:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
                ->first();
```

groupBy / have

Les méthodes `groupBy` et `having` peuvent être utilisées pour regrouper les résultats de la requête. La `having` signature de la méthode est similaire à celle de la `where` méthode:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

Vous pouvez transmettre plusieurs arguments à la `groupBy` méthode pour regrouper plusieurs colonnes:

```
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

Pour des `having` instructions plus avancées , voir la `havingRaw` méthode.

sauter / prendre

Pour limiter le nombre de résultats renvoyés par la requête ou pour ignorer un nombre donné de résultats dans la requête, vous pouvez utiliser les méthodes `skip` et `take` :

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternativement, vous pouvez utiliser les méthodes `limit` et `offset` :

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

Clauses conditionnelles

Parfois, vous pouvez souhaiter que les clauses s'appliquent à une requête uniquement lorsque quelque chose d'autre est vrai. Par exemple, vous souhaiterez peut-être appliquer une `where` instruction uniquement si une valeur d'entrée donnée est présente dans la demande entrante. Vous pouvez accomplir cela en utilisant la `when` méthode:

```
$role = $request->input('role');

$users = DB::table('users')
```

```
                ->when($role, function ($query, $role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

La `when` méthode n'exécute la fermeture donnée que lorsque le premier paramètre est `true`. Si le premier paramètre est `false`, la fermeture ne sera pas exécutée.

Vous pouvez passer une autre fermeture comme troisième paramètre de la `when` méthode. Cette fermeture sera exécutée si le premier paramètre est évalué en tant que `false`. Pour illustrer l'utilisation de cette fonctionnalité, nous allons l'utiliser pour configurer le tri par défaut d'une requête:

```
$sortBy = null;

$users = DB::table('users')
                ->when($sortBy, function ($query, $sortBy) {
                    return $query->orderBy($sortBy);
                }, function ($query) {
                    return $query->orderBy('name');
                })
                ->get();
```

#Inserts

Le générateur de requêtes fournit également une `insert` méthode pour insérer des enregistrements dans la table de base de données. La `insert` méthode accepte un tableau de noms de colonnes et de valeurs:

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

Vous pouvez même insérer plusieurs enregistrements dans la table avec un seul appel `insert` en passant un tableau de tableaux. Chaque tableau représente une ligne à insérer dans la table:

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

ID auto-incrémentés

Si la table a un identifiant auto-incrémenté, utilisez la `insertGetId` méthode pour insérer un enregistrement, puis récupérez l'identifiant:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

> Lors de l'utilisation de PostgreSQL, la `insertGetId` méthode attend que la colonne auto-incrémentée soit nommée `id`. Si vous souhaitez récupérer l'ID à partir d'une "séquence" différente, vous pouvez transmettre le nom de la colonne en tant que deuxième paramètre à la `insertGetId` méthode.

Mises à jour

Bien entendu, en plus d'insérer des enregistrements dans la base de données, le générateur de requêtes peut également mettre à jour des enregistrements existants à l'aide de la `update` méthode. La `update` méthode, comme la `insert` méthode, accepte un tableau de paires de colonnes et de valeurs contenant les colonnes à mettre à jour. Vous pouvez contraindre la `update` requête à l'aide de `where` clauses:

```
DB::table('users')
            ->where('id', 1)
            ->update(['votes' => 1]);
```

Mise à jour des colonnes JSON

Lors de la mise à jour d'une colonne JSON, vous devez utiliser la syntaxe pour accéder à la clé appropriée dans l'objet JSON. Cette opération est uniquement prise en charge sur MySQL 5.7+: `->`

```
DB::table('users')
            ->where('id', 1)
            ->update(['options->enabled' => true]);
```

Incrément & Décrément

Le générateur de requêtes fournit également des méthodes pratiques pour incrémenter ou décrémenter la valeur d'une colonne donnée. C'est un raccourci, fournissant une interface plus expressive et concise par rapport à l'écriture manuelle de l' `update` instruction.

Ces deux méthodes acceptent au moins un argument: la colonne à modifier. Un deuxième argument peut éventuellement être passé pour contrôler la quantité par laquelle la colonne doit être incrémentée ou décrémentée:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

Vous pouvez également spécifier des colonnes supplémentaires à mettre à jour pendant l'opération:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

#Supprime

Le constructeur de requêtes peut également être utilisé pour supprimer des enregistrements de la table via la `delete` méthode. Vous pouvez contraindre des `delete` instructions en ajoutant des `where` clauses avant d'appeler la `delete` méthode:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

Si vous souhaitez tronquer la table entière, ce qui supprimera toutes les lignes et réinitialisera l'ID d'auto-incrémentation à zéro, vous pouvez utiliser la `truncate` méthode suivante:

```
DB::table('users')->truncate();
```

#Verrouillage pessimiste

Le générateur de requêtes inclut également quelques fonctions pour vous aider à "verrouiller de manière pessimiste" vos `select` déclarations. Pour exécuter l'instruction avec un "verrou partagé", vous pouvez utiliser la `sharedLock` méthode sur une requête. Un verrou partagé empêche les lignes sélectionnées d'être modifiées jusqu'à ce que votre transaction soit validée:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

Sinon, vous pouvez utiliser la `lockForUpdate` méthode. Un verrou "pour la mise à jour" empêche les lignes d'être modifiées ou d'être sélectionnées avec un autre verrou partagé:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```