Database: Query Builder

Introduction

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works on all supported database systems.

The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean strings being passed as bindings.

PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns, etc. If you must allow the user to select certain columns to query against, always validate the column names against a white-list of allowed columns.

Retrieving Results

Retrieving All Rows From A Table

You may use the `table` method on the `DB` facade to begin a query. The `table` method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally get the results using the `get` method:

```php
<?php

namespace App\Http\Controllers;

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

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]);
    }
}
```

The `get` method returns an `Illuminate\Support\Collection` containing the results

where each result is an instance of the PHP `stdClass` object. You may access each column's value by accessing the column as a property of the object:

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

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from the database table, you may use the `first` method. This method will return a single `stdClass` object:

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

echo $user->name;
```

If you don't even need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

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

To retrieve a single row by its `id` column value, use the `find` method:

```
$user = DB::table('users')->find(3);
```

Retrieving A List Of Column Values

If you would like to retrieve a Collection containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve a Collection of role titles:

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

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

You may also specify a custom key column for the returned Collection:

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

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

Chunking Results

If you need to work with thousands of database records, consider using the `chunk` method. This method retrieves a small chunk of the results at a time and feeds each chunk into a `Closure` for processing. This method is very useful for writing Artisan commands that process thousands of records. For example, let's work with the entire `users` table in chunks of 100 records at a time:

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

You may stop further chunks from being processed by returning `false` from the `Closure`:

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

    return false;
});
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. So, when updating records while chunking, it is always best to use the `chunkById` method instead. This method will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
```

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

When updating or deleting records inside the chunk callback, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the chunked results.

Aggregates

The query builder also provides a variety of aggregate methods such as `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

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

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

You may combine these methods with other clauses:

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

Determining If Records Exist

Instead of using the `count` method to determine if any records exist that match your query's constraints, you may use the `exists` and `doesntExist` methods:

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

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

Selects

Specifying A Select Clause

You may not always want to select all columns from a database table. Using the `select` method, you can specify a custom `select` clause for the query:

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

The `distinct` method allows you to force the query to return distinct results:

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

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the `addSelect` method:

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

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

Raw Expressions

Sometimes you may need to use a raw expression in a query. To create a raw expression, you may use the `DB::raw` method:

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

Raw statements will be injected into the query as strings, so you should be extremely careful to not create SQL injection vulnerabilities.

Raw Methods

Instead of using `DB::raw`, you may also use the following methods to insert a raw expression into various parts of your query.

`selectRaw`

The `selectRaw` method can be used in place of `addSelect(DB::raw(...))`. This method accepts an optional array of bindings as its second argument:

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

`whereRaw` / `orWhereRaw`

The `whereRaw` and `orWhereRaw` methods can be used to inject a raw `where` clause into your query. These methods accept an optional array of bindings as their second argument:

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

`havingRaw` / `orHavingRaw`

The `havingRaw` and `orHavingRaw` methods may be used to set a raw string as the value of the `having` clause. These methods accept an optional array of bindings as their second argument:

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

`orderByRaw`

The `orderByRaw` method may be used to set a raw string as the value of the `order by` clause:

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

Joins

Inner Join Clause

The query builder may also be used to write join statements. To perform a basic "inner join", you may use the `join` method on a query builder instance. The first argument passed to the `join` method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. You can even join to multiple tables in a single query:

```
$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();
```

Left Join / Right Join Clause

If you would like to perform a "left join" or "right join" instead of an "inner join", use the `leftJoin` or `rightJoin` methods. These methods have the same signature as the `join` method:

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

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

Cross Join Clause

To perform a "cross join" use the `crossJoin` method with the name of the table you

wish to cross join to. Cross joins generate a cartesian product between the first table and the joined table:

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

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a `Closure` as the second argument into the `join` method. The `Closure` will receive a `JoinClause` object which allows you to specify constraints on the `join` clause:

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

If you would like to use a "where" style clause on your joins, you may use the `where` and `orWhere` methods on a join. Instead of comparing two columns, these methods will compare the column against a value:

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

Sub-Query Joins

You may use the `joinSub`, `leftJoinSub`, and `rightJoinSub` methods to join a query to a sub-query. Each of these methods receive three arguments: the sub-query, its table alias, and a Closure that defines the related columns:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as last_post_creat
                   ->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();
```

Unions

The query builder also provides a quick way to "union" two queries together. For example, you may create an initial query and use the `union` method to union it with a second query:

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

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

The `unionAll` method is also available and has the same method signature as `union`.

Where Clauses

Simple Where Clauses

You may use the `where` method on a query builder instance to add `where` clauses to the query. The most basic call to `where` requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. Finally, the third argument is the value to evaluate against the column.

For example, here is a query that verifies the value of the "votes" column is equal to 100:

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

For convenience, if you want to verify that a column is equal to a given value, you may pass the value directly as the second argument to the `where` method:

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

You may use a variety of other operators when writing a `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();
```

You may also pass an array of conditions to the `where` function:

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

Or Statements

You may chain where constraints together as well as add `or` clauses to the query. The `orWhere` method accepts the same arguments as the `where` method:

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

Additional Where Clauses

whereBetween / orWhereBetween

The `whereBetween` method verifies that a column's value is between two values:

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

whereNotBetween / orWhereNotBetween

The `whereNotBetween` method verifies that a column's value lies outside of two values:

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

whereIn / whereNotIn / orWhereIn / orWhereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

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

The `whereNotIn` method verifies that the given column's value is **not** contained in the given array:

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

whereNull / whereNotNull / orWhereNull / orWhereNotNull

The `whereNull` method verifies that the value of the given column is `NULL`:

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

The `whereNotNull` method verifies that the column's value is not `NULL`:

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

whereDate / whereMonth / whereDay / whereYear / whereTime

The `whereDate` method may be used to compare a column's value against a date:

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

The `whereMonth` method may be used to compare a column's value against a specific month of a year:

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

The `whereDay` method may be used to compare a column's value against a specific day of a month:

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

The `whereYear` method may be used to compare a column's value against a specific year:

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

The `whereTime` method may be used to compare a column's value against a specific time:

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

whereColumn / orWhereColumn

The `whereColumn` method may be used to verify that two columns are equal:

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

You may also pass a comparison operator to the method:

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

The `whereColumn` method can also be passed an array of multiple conditions. These conditions will be joined using the `and` operator:

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

Parameter Grouping

Sometimes you may need to create more advanced where clauses such as "where exists" clauses or nested parameter groupings. The Laravel query builder can handle these as well. To get started, let's look at an example of grouping constraints within parenthesis:

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

As you can see, passing a `Closure` into the `where` method instructs the query builder to begin a constraint group. The `Closure` will receive a query builder instance which you can use to set the constraints that should be contained within the parenthesis group. The example above will produce the following SQL:

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

You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

Where Exists Clauses

The `whereExists` method allows you to write `where exists` SQL clauses. The `whereExists` method accepts a `Closure` argument, which will receive a query builder instance allowing you to define the query that should be placed inside of the "exists" clause:

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

The query above will produce the following SQL:

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

JSON Where Clauses

Laravel also supports querying JSON column types on databases that provide support for JSON column types. Currently, this includes MySQL 5.7, PostgreSQL, SQL Server 2016, and SQLite 3.9.0 (with the JSON1 extension). To query a JSON column, use the `->` operator:

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

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

You may use `whereJsonContains` to query JSON arrays (not supported on SQLite):

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

MySQL and PostgreSQL support `whereJsonContains` with multiple values:

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

```
                ->get();
```

You may use `whereJsonLength` to query JSON arrays by their length:

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

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

Ordering, Grouping, Limit, & Offset

orderBy

The `orderBy` method allows you to sort the result of the query by a given column. The first argument to the `orderBy` method should be the column you wish to sort by, while the second argument controls the direction of the sort and may be either `asc` or `desc`:

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

latest / oldest

The `latest` and `oldest` methods allow you to easily order results by date. By default, result will be ordered by the `created_at` column. Or, you may pass the column name that you wish to sort by:

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

inRandomOrder

The `inRandomOrder` method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

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

groupBy / having

The `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

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

You may pass multiple arguments to the `groupBy` method to group by multiple columns:

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

For more advanced `having` statements, see the `havingRaw` method.

skip / take

To limit the number of results returned from the query, or to skip a given number of results in the query, you may use the `skip` and `take` methods:

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

Alternatively, you may use the `limit` and `offset` methods:

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

```
                ->get();
```

Conditional Clauses

Sometimes you may want clauses to apply to a query only when something else is true. For instance you may only want to apply a `where` statement if a given input value is present on the incoming request. You may accomplish this using the `when` method:

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

$users = DB::table('users')
                ->when($role, function ($query, $role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

The `when` method only executes the given Closure when the first parameter is `true`. If the first parameter is `false`, the Closure will not be executed.

You may pass another Closure as the third parameter to the `when` method. This Closure will execute if the first parameter evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default sorting of a query:

```
$sortBy = null;

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

Inserts

The query builder also provides an `insert` method for inserting records into the database table. The `insert` method accepts an array of column names and values:

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

You may even insert several records into the table with a single call to `insert` by passing an array of arrays. Each array represents a row to be inserted into the table:

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

The `insertOrIgnore` method will ignore duplicate record errors while inserting records into the database:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'taylor@example.com'],
    ['id' => 2, 'email' => 'dayle@example.com']
]);
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

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

When using PostgreSQL the `insertGetId` method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the column name as the second parameter to the `insertGetId` method.

Updates

In addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs containing the columns to be updated. You may constrain the `update` query using `where` clauses:

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

Update Or Insert

Sometimes you may want to update an existing record in the database or create it if no matching record exists. In this scenario, the `updateOrInsert` method may be used. The `updateOrInsert` method accepts two arguments: an array of conditions by which to find the record, and an array of column and value pairs containing the columns to be updated.

The `updateOrInsert` method will first attempt to locate a matching database record using the first argument's column and value pairs. If the record exists, it will be updated with the values in the second argument. If the record can not be found, a new record will be inserted with the merged attributes of both arguments:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

Updating JSON Columns

When updating a JSON column, you should use `->` syntax to access the appropriate key in the JSON object. This operation is supported on MySQL 5.7+ and PostgreSQL 9.5+:

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

Increment & Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. This is a shortcut, providing a more expressive and terse interface compared to manually writing the `update` statement.

Both of these methods accept at least one argument: the column to modify. A second argument may optionally be passed to control the amount by which the column should be incremented or decremented:

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

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

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

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

You may also specify additional columns to update during the operation:

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

Deletes

The query builder may also be used to delete records from the table via the `delete` method. You may constrain `delete` statements by adding `where` clauses before calling the `delete` method:

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

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

If you wish to truncate the entire table, which will remove all rows and reset the auto-incrementing ID to zero, you may use the `truncate` method:

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

Pessimistic Locking

The query builder also includes a few functions to help you do "pessimistic locking" on your `select` statements. To run the statement with a "shared lock", you may use the `sharedLock` method on a query. A shared lock prevents the selected rows from being modified until your transaction commits:

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

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the rows from being modified or from being selected with another shared lock:

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

Debugging

You may use the `dd` or `dump` methods while building a query to dump the query bindings and SQL. The `dd` method will display the debug information and then stop executing the request. The `dump` method will display the debug information but allow the request to keep executing:

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

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

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.