# Database: Query Builder

## # Introduction

The 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.

**Note:** 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.

## # Retrieving Results

### Retrieving All Rows From A Table

To begin a fluent query, use the `table` method on the `DB` facade. 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. In this example, let's just `get` all records from a table:

```php
<?php

namespace App\Http\Controllers;

use 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]);
    }
}
```

Like raw queries, the `get` method returns an `array` of 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:

```php
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:

```php
$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:

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

### Chunking Results From A Table

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')->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')->chunk(100, function($users) {
    // Process the records...

    return false;
});
```

### Retrieving A List Of Column Values

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

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

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

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

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

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

## 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');
```

Of course, you may combine these methods with other clauses to build your query:

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

## # Selects

### Specifying A Select Clause

Of course, 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. These expressions will be injected into the query as strings, so be careful not to create any SQL injection points! 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();
```

## # Joins

### Inner Join Statement

The query builder may also be used to write join statements. To perform a basic SQL "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. Of course, as you can see, you can 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 Statement

If you would like to perform a "left join" instead of an "inner join", use the `leftJoin` method. The `leftJoin` method has the same signature as the `join` method:

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

### Advanced Join Statements

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();
```

## # Unions

The query builder also provides a quick way to "union" two queries together. For example, you may create an initial query, and then 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

To add `where` clauses to the query, use the `where` method on a query builder instance. 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. 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 simply 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();
```

Of course, 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();
```

### 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

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

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

#### whereNotBetween

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

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

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();
```

## # Advanced Where Clauses

### Parameter Grouping

Sometimes you may need to create more advanced where clauses such as "where exists" 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:

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

As you can see, passing `Closure` into the `orWhere` 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' or (votes > 100 and title <> 'Admin')
```

### Exists Statements

The `whereExists` method allows you to write `where exist` 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:

```
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
)
```

## # 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();
```

### groupBy / having / havingRaw

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();
```

The `havingRaw` method may be used to set a raw string as the value of the `having` clause. For example, we can find all of the departments with sales greater than $2,500:

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

### skip / take

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

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

```
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]
]);
```

### 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]
);
```

**Note:** 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 sequence name as the second parameter to the `insertGetId` method.

## # Updates

Of course, 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:

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

### Increment / Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. This is simply a short-cut, 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 / 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

Of course, the query builder may also be used to delete records from the table via the `delete` method:

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

You may constrain `delete` statements by adding `where` clauses before calling the `delete` method:

```
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();
```