# 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 perfectly with all of Laravel's 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 or sanitize strings passed to the query builder as query bindings.

# Retrieving All Rows From A Table

You may use the table method provided by 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 retrieve the results of the query using the get method:

<?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 \Illuminate\Http\Response

\*/

public function index()

{

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

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

}

}

# Retrieving A Single Row / Column From A Table

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

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

return $user->email;

If you don't 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 an Illuminate\Support\Collection instance containing the values of a single column, you may use the pluck method. In this example, we'll retrieve a collection of user titles:

use Illuminate\Support\Facades\DB;

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

foreach ($titles as $title) {

echo $title;

}

You may specify the column that the resulting collection should use as its keys by providing a second argument to the pluck method:

$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {

echo $title;

}

# Aggregates

The query builder also provides a variety of methods for retrieving aggregate values like count, max, min, avg, and sum. You may call any of these methods after constructing your query:

use Illuminate\Support\Facades\DB;

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

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

Of course, you may combine these methods with other clauses to fine-tune how your aggregate value is calculated:

$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:

if (DB::table('orders')->where('finalized', 1)->exists()) {

// ...

}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {

// ...

}

# Select Statements

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:

use Illuminate\Support\Facades\DB;

$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();

# Joins

## Inner Join Clause

The query builder may also be used to add join clauses to your queries. 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 may even join multiple tables in a single query:

use Illuminate\Support\Facades\DB;

$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

You may use the crossJoin method to perform a "cross join". Cross joins generate a cartesian product between the first table and the joined table:

$sizes = DB::table('sizes')

->crossJoin('colors')

->get();

# Basic Where Clauses

## Where Clauses

You may use the query builder's where method to add "where" clauses to the query. The most basic call to the where method 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 compare against the column's value.

For example, the following query retrieves users where the value of the votes column is equal to 100 and the value of the age column is greater than 35:

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

->where('votes', '=', 100)

->where('age', '>', 35)

->get();

For convenience, if you want to verify that a column is = to a given value, you may pass the value as the second argument to the where method. Laravel will assume you would like to use the = operator:

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

As previously mentioned, you may use any operator that is supported by your database system:

$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. Each element of the array should be an array containing the three arguments typically passed to the where method:

$users = DB::table('users')->where([

['status', '=', '1'],

['subscribed', '<>', '1'],

])->get();

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.

## Or Where Clauses

When chaining together calls to the query builder's where method, the "where" clauses will be joined together using the and operator. However, you may use the orWhere method to join a clause to the query using the or operator. The orWhere method accepts the same arguments as the where method:

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

->where('votes', '>', 100)

->orWhere('name', 'John')

->get();

If you need to group an "or" condition within parentheses, you may pass a closure as the first argument to the orWhere method:

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

->where('votes', '>', 100)

->orWhere(function($query) {

$query->where('name', 'Abigail')

->where('votes', '>', 50);

})

->get();

The example above will produce the following SQL:

select \* from users where votes > 100 or (name = 'Abigail' and votes > 50)

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

## Where Not Clauses

The whereNot and orWhereNot methods may be used to negate a given group of query constraints. For example, the following query excludes products that are on clearance or which have a price that is less than ten:

$products = DB::table('products')

->whereNot(function ($query) {

$query->where('clearance', true)

->orWhere('price', '<', 10);

})

->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();

If you are adding a large array of integer bindings to your query, the whereIntegerInRaw or whereIntegerNotInRaw methods may be used to greatly reduce your memory usage.

## 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:

$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 the 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 whereColumn method:

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

->whereColumn('updated\_at', '>', 'created\_at')

->get();

You may also pass an array of column comparisons to the whereColumn method. These conditions will be joined using the and operator:

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

->whereColumn([

['first\_name', '=', 'last\_name'],

['updated\_at', '>', 'created\_at'],

# Ordering, Grouping, Limit & Offset

## Ordering

### The orderBy Method

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

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

->orderBy('name', 'desc')

->get();

To sort by multiple columns, you may simply invoke orderBy as many times as necessary:

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

->orderBy('name', 'desc')

->orderBy('email', 'asc')

->get();

### The latest & oldest Methods

The latest and oldest methods allow you to easily order results by date. By default, the result will be ordered by the table's created\_at column. Or, you may pass the column name that you wish to sort by:

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

->latest()

->first();

### Random Ordering

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();

### Removing Existing Orderings

The reorder method removes all of the "order by" clauses that have previously been applied to the query:

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

$unorderedUsers = $query->reorder()->get();

You may pass a column and direction when calling the reorder method in order to remove all existing "order by" clauses and apply an entirely new order to the query:

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

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();

## Grouping

### The groupBy & having Methods

As you might expect, 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 can use the havingBetween method to filter the results within a given range:

$report = DB::table('orders')

->selectRaw('count(id) as number\_of\_orders, customer\_id')

->groupBy('customer\_id')

->havingBetween('number\_of\_orders', [5, 15])

->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();

To build more advanced having statements, see the havingRaw method.

## Limit & Offset

### The skip & take Methods

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

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

Alternatively, you may use the limit and offset methods. These methods are functionally equivalent to the take and skip methods, respectively:

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

->offset(10)

->limit(5)

->get();

# Insert Statements

The query builder also provides an insert method that may be used to insert records into the database table. The insert method accepts an array of column names and values:

DB::table('users')->insert([

'email' => 'kayla@example.com',

'votes' => 0

]);

You may insert several records at once by passing an array of arrays. Each array represents a record that should be inserted into the table:

DB::table('users')->insert([

['email' => 'picard@example.com', 'votes' => 0],

['email' => 'janeway@example.com', 'votes' => 0],

]);

## insertOrIgnore

The insertOrIgnore method will ignore errors while inserting records into the database. When using this method, you should be aware that duplicate record errors will be ignored and other types of errors may also be ignored depending on the database engine. For example, insertOrIgnore will bypass MySQL's strict mode:

DB::table('users')->insertOrIgnore([

['id' => 1, 'email' => 'sisko@example.com'],

['id' => 2, 'email' => 'archer@example.com'],

]);

## insertUsing

The insertUsing method will insert new records into the table while using a subquery to determine the data that should be inserted:

DB::table('pruned\_users')->insertUsing([

'id', 'name', 'email', 'email\_verified\_at'

], DB::table('users')->select(

'id', 'name', 'email', 'email\_verified\_at'

)->where('updated\_at', '<=', now()->subMonth()));

## 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]

);

# Update Statements

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 indicating the columns to be updated. The update method returns the number of affected rows. 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 indicating the colaumns to be updated.

The updateOrInsert method will 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 update 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. Both of these methods accept at least one argument: the column to modify. A second argument may be provided to specify 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']);

# Delete Statements

The query builder's delete method may be used to delete records from the table. The delete method returns the number of affected rows. You may constrain delete statements by adding "where" clauses before calling the delete method:

$deleted = DB::table('users')->delete();

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

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

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

## Eloquent vs Query Builder and Why

Yes, In some case you are right. When we've more data and almost in every site, data is not small really. Then it is better to use DB Query than the Eloquent Query.

## So Why then Eloquent? Isn't any necessary that?

Answer is - Eloquent is also necessary. Cause-

Yes, In some case you are right. When we've more data and almost in every site, data is not small really. Then it is better to use DB Query than the Eloquent Query.

In a performance issue of Eloquent VS DB I've heard that,

To insert 1000 rows for a simple table Eloquent takes 1.2 seconds and in that case DB facades take only 800 mili seconds(ms).

To create a better relationship and get the results in view with so much simple syntax, when there needs to join.

Eloquent is also for who have not much knowledge of SQL query.

An MVC framework follow the rules of Code Readability, Code Maintainability and which Eloquent is, you know that. A code comparison below. Obviously, Eloquent is better to read.

Most important part is if we want to change other database, then raw query will be a lot much headache to us and in that case Laravel Eloquent will solve all the problems with one hand. It can handle different types Database.

## Real Life Example

1. I'm making a university website. Which may contain maximum 5,000

teachers and 10,000 students and some notices and files. Then it is

better to do this with simple Laravel Eloquent which is very much

standard and readable.

2. Now I'm making a site like Stack overflow. Which may contain more

than 1,000,0000 (1 crore) posts and many more things. I must

choose the conventional DB facades there. It is faster for searching the

posts from so many records.