Database: Getting Started

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 All Rows from a Table	`DB::table('users')->get();`

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

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:

```
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.	`DB::table('users')->where('name', 'John')->value('email');`
To retrieve a single row by its `id` column value, use the `find` method	`DB::table('users')->find(3);`

Retrieving a List of Column Values	DB::table('roles')->pluck('title');

If you would like to retrieve a Collection containing the values of a single column, you may use the pluck method.	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.

```
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]);
        }
    });
```

Aggregates	The query builder also provides a variety of aggregate methods such as count, max, min, avg, and sum.
DB::table('users')->count();	DB::table('orders')->max('price');
DB::table('orders')->where('finalized', 1)->avg('price');	
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	
DB::table('orders')->where('finalized', 1)->exists();	DB::table('orders')->where('finalized', 1)->doesntExist();
Selects	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
DB::table('users')->select('name', 'email as user_email')->get();	The distinct method allows you to force the query to return distinct results: 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();
Sometimes you may need to use a raw expression in a query. To create a raw expression, you may use the DB::raw method. Raw statements will be injected into the query as strings, so you should be extremely careful to not create SQL injection vulnerabilities.	DB::table('users') ->select(DB::raw('count(*) as user_count, status')) ->where('status', '<>', 1) ->groupBy('status') ->get();
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.
The selectRaw method can be used in place of addSelect(DB::raw(...)). This method accepts an optional array of bindings as its second argument:	DB::table('orders')->selectRaw('price * ? as price_with_tax', [1.0825])->get();
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	DB::table('orders')->whereRaw('price > IF(state = "TX", ?, 100)', [200]) ->get()

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.	DB::table('orders') ->select('department', DB::raw('SUM(price) as total_sales')) ->groupBy('department') ->havingRaw('SUM(price) > ?', [2500]) ->get();
The orderByRaw method may be used to set a raw string as the value of the order by clause.	DB::table('orders')->orderByRaw('updated_at - created_at DESC')->get();
The groupByRaw method may be used to set a raw string as the value of the group by clause.	DB::table('orders')->select('city', 'state')->groupByRaw('city, state') ->get();
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
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
DB::table('users') ->leftJoin('posts', 'users.id', '=', 'posts.user_id') ->get();	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
DB::table('sizes')->crossJoin('colors')->get();	
Advanced Join Clauses	
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) { $join->on('users.id', '=', 'contacts.user_id') ->where('contacts.user_id', '>', 5); })->get();
Subquery Joins	You may use the joinSub, leftJoinSub, and rightJoinSub methods to join a query to a subquery. Each of these methods receive three arguments: the subquery, its table alias, and a Closure that defines the related columns
DB::table('posts')->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))->where('is_published', true) ->groupBy('user_id');	
DB::table('users')->joinSub($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.	
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.
DB::table('users')->where('votes', '=', 100)->get();	DB::table('users')->where('votes', 100)->get();
DB::table('users')->where('votes', '>=', 100)->get();	DB::table('users')->where('votes', '<>', 100)->get();
DB::table('users')->where('name', 'like', 'T%')->get();	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

DB::table('users')->where('votes', '>', 100)
->orWhere('name', 'John')->get();
	DB::table('users')->where('votes', '>', 100)->orWhere(function($query) {
	$query->where('name', 'Abigail')
	->where('votes', '>', 50);
	})->get();
	// SQL: select * from users where votes > 100 or (name = 'Abigail' and votes > 50)

whereBetween / orWhereBetween	The whereBetween method verifies that a column's value is between two values:
DB::table('users')->whereBetween('votes', [1, 100])->get();	
whereNotBetween / orWhereNotBetween	The whereNotBetween method verifies that a column's value lies outside of two values:
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	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	DB::table('users')->whereNotIn('id', [1, 2, 3])->get(); If you are adding a huge 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:	`DB::table('users')->whereNull('updated_at')->get();`
The whereNotNull method verifies that the column's value is not NULL:	`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:	`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:	`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	`DB::table('users')->whereDay('created_at', '31')->get();`
The whereYear method may be used to compare a column's value against a specific year	`DB::table('users')->whereYear('created_at', '2016')->get();`
The whereTime method may be used to compare a column's value against a specific time	`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:	`DB::table('users')->whereColumn('first_name', 'last_name')->get();`

You may also pass a comparison operator to the method	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	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

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

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

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

Subquery Where Clauses	Sometimes you may need to construct a where clause that compares the results of a subquery to a given value. You may accomplish this by passing a Closure and a value to the where method. For example, the following query will retrieve all users who have a recent "membership" of a given type

```
User::where(function ($query) { $query->select('type')
        ->from('membership')
        ->whereColumn('user_id', 'users.id')
        ->orderByDesc('start_date')
        ->limit(1);
}, 'Pro')->get();
```

Ordering, Grouping, Limit & Offset	
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	DB::table('users')->orderBy('name', 'desc')->get();
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();
The inRandomOrder method may be used to sort the query results randomly. For example, you may use this method to fetch a random user	DB::table('users')->inRandomOrder()->first();
The reorder method allows you to remove all the existing orders and optionally apply a new order. For example, you can remove all the existing orders	$query = DB::table('users')->orderBy('name'); $unorderedUsers = $query->reorder()->get();
To remove all existing orders and apply a new order, provide the column and direction as arguments to the method	$query = DB::table('users')->orderBy('name'); $usersOrderedByEmail = $query->reorder('email', 'desc')->get();
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	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:	DB::table('users')->groupBy('first_name', 'status') ->having('account_id', '>', 100)->get();

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	DB::table('users')->skip(10)->take(5)->get(); DB::table('users')->offset(10)->limit(5)->get();

Inserts	DB::table('users')->insert(['email' => 'john@example.com', 'votes' => 0]); 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']]);
Updates	DB::table('users')->where('id', 1)->update(['votes' => 1]);
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']);
Deletes	DB::table('users')->delete(); DB::table('users')->where('votes', '>', 100)->delete();
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();

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();