Laravel where not null

In this tutorial we learn how to ask in Laravel where zero and where not zero. I will use the raw query and convert it to laravel eloquent. Suppose we have a table named Post with the following values. Post Table Id title post_description 1 test title this is test description 2 null this is 2nd test description 3 2nd title null Where Not Null Query SQL raw query Select * from posts where title IS NOT NULL; Laravel eloquent query DB::table('posts') ->whereNotNull('title') ->get(); Where Null Query sql raw query * is selected from posts where the title is null; Laravel eloquent query DB::table('posts') ->whereNull('title') ->get(); If you are a starter or learner laravel, then you have in mind how to check where zero or where not zero condition with laravel eloquent model if you write eloquent query. When you write a query for SQL, it is very simple syntax and families with where zero and not zero condition. In this little tutorial I will let you know how to write query where zero and where not zero condition. here I will write core SQL query and then convert to laravel query builder with db. So just see example below and see how it works.we have the following solution on SQL and Laravel query builder:1) IS NULL = whereNull()2) IS NOL NULL = whereNotNull()Where null Query:SQL Query:SELECT * FROM users WHERE name IS NULL; Laravel Query:DB::table('users') ->whereNull('name') ->get(); Where Not Null Query:SQL Query:SELECT * FROM users WHERE name IS NOT NULL; Laravel Query:DB::table('users') ->whereNotNull('name') ->get();I hope you have found your best solution.... 2 April 2018 | Category : PHPLaravelMySQLSQLLaravel 5Laravel 5.5Laravel 5.6 Posts © Laracasts 2020. All rights reserved. Yes, everyone. That is, you, Todd. Designed with by Tuds. Proudly hosted with Laravel Forge and DigitalOcean. Introducing Laravel's Database Query Builder provides a convenient, fluid interface for 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 from SQL injection attacks. There is no need to clean strings that are passed as bindings. Note 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, and so on. If you need to allow the user to select specific columns for which you want to query, always check the column names against a List of allowed columns. Get results Get all rows from a table You can use the table method on the DB facade to start a query. The table method returns a flowing query builder instance for the specified table, so that you can chain further constraints on the query, and then finally the with the get method: <?php Namespace app- Use the app,http controller and controller; Use Illuminate Support Facade db; Class UserController Extends Controller -/** * Display a list of all users of the application. * * @return answer */ Public Function index() - $users = DB::table('users')->get(); return view('user.index', ['users' => $users]); The get method returns an Illuminate Support collection that contains the results where each result is an instance of the PHP stdClass object. You can access the value of each column by accessing the column as a property of the object: foreach ($users as $user) - Echo $user->name; Get a single row/column from a table If you only need to retrieve a single row from the database table, you can use the first method. This method returns 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 can extract a single value from a record using the Value method. This method returns the value of the column directly: $email = DB::table('users')->where('name', 'John')->value('email'); To retrieve a single row based on its ID column value, use the find method: $user = DB::table('users')->find(3); Get a list of column values If you want to retrieve a collection that contains the values of a single column, you can use the plucking method. In this example, we get a collection of role titles: $titles = DB::table('roles')->pluck('title'); for everyone ($titles as $title) - Echo $title; You can also specify a custom key column for the returned collection: $roles = DB::table('roles')->pluck('title', 'name'); for each ($roles such as $name => $title $title) chunking results If you need to work with thousands of database records, you should use the Chunk method. This method retrieves a small portion of the results and feeds each chunk into a closure for processing. This method is very useful for writing Artisan commands that process thousands of records. For example.B. work with the entire user table in blocks of 100 records simultaneously: DB::table('users')->orderBy('id')->chunk(100, function ($users) ' foreach ($users as $user) ' . You can prevent further blocks from being processed by returning false from the closure: DB::table('users')->orderBy('id')->chunk(100, function ($users) ' / Process the records back... Return incorrectly; }); If you update database records as the results are splitting, the results of the block may be unexpected So, when you update records during chunking, it's always best to use the chunkById method instead. This method automatically paginates the results based on the primary key of the record: DB::table('users')->where('active', false) ->chunkById(100, function ($users) ' foreach ($users as $user) - DB::table('users') ->where('id', $user->id) } }); Note When updating or deleting records within the chunk callback, any changes to the primary key or foreign keys can affect the chunk query. This can result in records not being included in the blocked results. Aggregates The Query Builder also provides a variety of aggregate methods such as Count, max, min, avg, and sum. You can call one of these methods after you create your query: $users = DB::table('users')->count(); $price = DB::table('orders')->max('price'); You can combine these methods with other clauses: $price = DB::table('orders') ->where('finalized', 1) ->avg('price'); To determine whether records exist instead of using the Count method to determine whether there are records that match the constraints of your query, you can use the exists and doesntExist methods: DB::table('orders')->where('finalized', 1)->exists(); DB::table('orders')->where('finalized', 1)->doesntExist(); Selects Specifya selection clause You may not always want to select all columns from a database table. The select method allows you to specify a custom selection clause for the query: $users = DB::table('users')->select('name', 'email as user_email'))->get(); The unique method allows you to force the query to return different results: $users = DB::table('users')->distinct()->get(); If you already have a query builder instance and want to add a column to the existing Select clause, you can use the addSelect method: $query = DB::table('users')->select('name'); $users = $query->addSelect('alter')->get(); Raw expressions Sometimes you may need to use a raw expression in a query. To create a raw expression, you can use the DB::raw method: $users = DB::table('users') ->select(DB::raw('count(*) as user_count, status')) ->where('status', '<>', 1) ->groupBy('status') ->get(); Note Raw statements are inserted as strings in the query, so be extremely careful not to create SQL injection vulnerabilities. Raw methods Instead of using DB::raw, you can also use the following methods to insert a raw expression into different parts of the query. selectRaw The selectRaw method can be used instead of addSelect(DB::raw(...)). This method accepts an optional array of bindings as the 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. This accept an optional array of bindings as a second argument: $orders = DB::table('orders') ->whereRaw('price > IF(state = TX, ?, 100)', [200]) ->get(); havingRaw / orHavingRaw The havingRaw and orHavingRaw methods can be used to set a raw string as the value of the having clause. These methods accept an optional array of bindings as the second argument: $orders = DB::table('orders') ->select('department', DB::raw('SUM(price) DB::raw('SUM(price) total_sales')) ->groupBy('department') ->havingRaw('SUM(price) > ?', [2500]) ->get(); orderByRaw The orderByRaw method can be used to set an raw string as the value of the order by clause: $orders = DB::table('orders') ->orderByRaw('updated_at - created_at DESC') ->get(); groupByRaw The groupByRaw method can be used to set a raw string as the value of the group by clause: $orders = DB::table('orders') ->select('city', 'state') ->groupByRaw('city, state') ->get(); Joins Inner Join Clause The Query Builder can also be used to write join statements. To perform a basic inner join, you can use the Join method on a query builder instance. The first argument passed to the join method is the name of the table that you must join, while the remaining arguments specify the column constraints for the join. You can even join 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 links join / Right Join Clause If you want to perform a left link or right link instead of an inner link, use the leftJoin right. 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 cross link, use the crossJoin method with the name of the table you want to cross the join with. Cross links generate a Cartesian product between the first table and the linked table: $sizes = DB::table('sizes') ->crossJoin('colors') ->get(); Advanced join clauses You can also specify extended join clauses. To get started, pass a closure as a second argument in the join method. The closure receives a JoinClause object that allows you to specify restrictions on the join clause: DB::table('users') ->join('contacts', function ($join) ' $join->on('users.id', '=', 'contacts.user_id')->orOn(...) >; If you want to use a Where style clause for your joins, you can use the Where and OrWhere methods for a join. Instead of comparing two columns, these methods compare the column to a value: DB::table('users') ->join('contacts', function ($join) ' $join->on('users.id', '=', 'contacts.user_id') ->where('contacts.user_id', '>', 5 >); You can use the joinSub, and rightJoinSub to connect a query to a subquery. Each of these methods has three arguments: the subquery, its table alias, and a closure that defines its columns: $latestPosts = DB::table('posts') ->select('user_id', DB::raw('CREATED_AT) as last_post_created_at')) ->where('is_published', true) ->groupBy('user_id'); $users = DB::table('users') ->joinSub($latestPosts, ->joinSub($latestPosts, function ($join) , $join->on('users.id', '=', 'latest_posts.user_id'); ->get(); Unions The Query Builder also provides a quick way to unionize two or more queries together. For example.B. you can create an initial query and use the Union method to merge it with additional queries: $first = DB::table('users') ->whereNull('first_name'); $users = DB::table('users') ->whereNull('last_name') ->union($first) ->get(); Stip The unionAll method is also available and has the same method signature as union. Where Clauses Simple Where Clauses You can use the where method in a query builder instance to add where clauses to the query. The simplest call where requires three arguments. The first argument is the name of the column. The second argument is an operator that can be any of the supported operators in the database. Finally, the third argument is the value to evaluate for the column. For example.B. is a query that checks whether the value of the votes column is 100: $users = DB::table('users')->where('votes', '=', 100)->get(); If you want to verify that a column is equal to a specific value, you can pass the value directly as a second argument to the where method: $users = DB::table('users')->where('votes', 100)->get(); You can use a variety of other operators when you write 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 can also pass a number of conditions to the where function: $users = DB::table('users')->where([['status', '=', '1'], ['subscribed', '<>', '1'],])->get(); Or instructions You can chain constraints together, add them to the query, or add clauses. 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 one or condition within the parentheses, you can pass a closure as the first argument to the orWhere method: $users = DB::table('users') ->('votes', '>', 100) ->orWhere($query) ' $query->where('name') ->where('votes', '>'>) SQL: 100 or (name = Abigail and > 50) Additional Where clauses that are between two values of a column: $users = DB::table('users') ->whereBetween('votes', [1, 100]) ->get(); whereNotBetween / orNotBetweenThe whereNot method checks whether the value of a column is outside of two values: $users = = ->whereNotBetween('votes', [1, 100]) ->get(); whereIn / whereNotIn / orWhereNot / orWhereNotIn The whereIn method checks whether the value of a particular column is contained in the specified array: $users = DB::table('users') ->whereIn('id', [1, 2, 3]) 3]) The whereNotIn method checks whether the value of the specified column is not in the specified array: $users = DB::table('users') ->whereNotIn('id', [1, 2, 3]) ->get(); Note If you add a large number of integer bindings to your query, the WhereIntegerInRaw or WhereIntegerNotInRaw methods can be used to significantly reduce memory usage. whereNull / whereNotNull / orWhereNotNull / orWhereNotNull The whereNull method checks whether the value of the specified column is null: $users = DB::table('users') ->whereNull('updated_at') ->get(); The whereNotNull method checks whether the value of the column is not null: $users = DB::table('users') ->whereNotNull('updated_at') ->get(); whereDate / whereMonth / whereDay / whereYear / whereTime The whereDate method can be used to compare the value of a column with a date: $users = DB::table('users') ->whereDate('created_at', '2016-12-31') ->get(); The whereMonth method can be used to compare the value of a column to a specific month of a year: $users = DB::table('users') ->whereMonth('created_at', '12') ->get(); The whereDay method can be used to compare the value of a column to a specific day of a month: $users = DB::table('users') ->whereDay('created_at', '31') ->get(); The whereYear method can be used to compare the value of a column to a specific year: $users = DB::table('users') ->whereYear('created_at', '2016') ->get(); The whereTime method can be used to compare the value of a column with a specific time: $users = DB::table('users') ->whereTime('created_at', '=', '11:20:45') ->get(); whereColumn / orWhereColumn The whereColumn method can be used to verify that two columns are equal: $users = DB::table('users') ->whereColumn('first_name', 'last_name') ->get(); You can 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 to an array of multiple conditions. These conditions are associated with 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 advanced clauses such as where there are clauses or nested parameter groupings. The Laravel query generator can also handle them. To begin, let's look at an example of grouping constraints within the bracket: $users = DB::table('users') ->where('name', '=', 'John') ->where($query) ' $query->where('votes', '>', 100) ->orWhere('title', "title", "', As you can see, passing a closure to the Where method is instructed to start a constraint group. Completion is given to a query generator instance that allows you to set the constraints to include in the bracket group. In the example above, the following SQL is generated: Select * from users where name = = and (Voices > 100 or Title = 'Admin') 'tip' You should always group orwhere calls to avoid unexpected behavior when global ranges are applied. Where Exists clauses The whereExists method allows you to write where SQL clauses exist. The whereExists method accepts a closure argument that receives a query builder instance that allows you to define the query to be placed within the exists clause: $users = DB::table('users') ->whereExists($query) - $query->select(DB::raw(1)) -> >from('orders') ->orders.user_id where The above query generates the following SQL: Select * from users where existing (choose 1 from jobs where orders.user_id = users.id) Sometimes you may need to create a where clause that compares the results of a subquery to a specific value. You can do this by passing a closure and value to the where method. For example.B, the following query retrieves all users who have a current membership of a particular type. Use app models users; $users = User::where($query) - $query->select('type') ->from('membership') ->whereColumn('user_id', 'users.id') ->orderByDesc('start_date') ->limit(1 >); JSON Where Clauses Laravel also supports querying JSON column types in databases that support 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 ->: $users = DB::table('users') ->where('options->language', 'en') ->get(); $users = DB::table('users') ->where('preferences->dining->meal', 'salad') ->get(); You can use whereJsonContains to queries 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 can use whereJsonLength to check JSON arrays for 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 specific column. The first argument to the orderBy method should be the column by which you want to sort, while the second argument controls the direction of the collation and can be either asc or desc: $users = DB::table('users') 'desc') ->get(); If you need to sort by multiple columns, you can call orderBy paintable as needed: $users = DB::table('users') ->orderBy('name', 'desc') ->orderBy('email', 'asc') ->get(); Latest/oldest The latest and oldest methods allow you to order results easily by date. By default, the result is sorted by column created_at. Or or can pass the column name by which you want to sort: $user = DB::table('users') ->latest() ->first(); inRandomOrder The inRandomOrder method can be used to sort the query results randomly. For example, you can use this method to retrieve a random user: $randomUser = DB::table('users') ->inRandomOrder() ->first(); Rearrange The reordering method allows you to remove all existing orders and optionally apply a new purchase order. You can remove .B all existing jobs: $query = DB::table('users')->orderBy('name'); $unorderedUsers = $query->reorder()->get(); To remove all existing jobs and apply a new order, specify the column and direction as arguments for the method: $query = DB::table('users')->orderBy('name'); $usersOrderedByEmail = $query->reorder('email', 'desc')->get(); groupBy / with the groupBy and with methods can be used to group the query results. The signature of the having method is similar to that of the where method: $users = DB::table('users') ->groupBy('account_id') ->having('account_id', '>', 100) ->get(); You can pass multiple arguments to the groupBy method to group them by multiple columns: $users = DB::table('users') ->groupBy('first_name', 'status') ->having('account_id', '>', 100) ->get(); For more information about advanced co-statements, see the havingRaw method. skip / take To limit the number of results returned by the query or to skip a certain number of results in the query, you can use the Skip methods: $users = DB::table('users')->skip(10)->take(5)->get(); Alternatively, you can use the limit and offset methods: $users = DB::table('users') ->offset(10) ->limit(5) ->get(); Conditional clauses Sometimes you want clauses to be applied to a query only if something else is true. For example, you can apply a where statement only if there is a given input value for the incoming request. You can do this using the when method: $role = $request->input('role'); $users = DB::table('users') ->when($role, function ($query, $role > $role role_id > $query) The when method executes the specified closure only if the first parameter is true. If the first parameter is false, the closure will not run. You can pass another closure as a third parameter to the when method. This closure is executed when the first parameter evaluates to false. To illustrate how this feature can be used, we use it to configure the default sorting of a query: $sortBy zero; $users = DB::table('users') ->when($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' => '', 'votes' => 0]); You can even use several multiple 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' => '', 'votes' => 0], ['email' => '', 'votes' => 0],]); The insertOrIgnore method ignores duplicate record errors when inserting records into the database: DB::table('users')->insertOrIgnore([['id' => 1, 'email' => ''], ['id' => 2, 'email' => ''] The upsert method inserts non-existent rows and updates the rows that already exist with the new values. The first argument of the method consists of the values to insert or update, while the second argument lists the columns that uniquely identify records within the associated table. The third and final argument of the method is an array of columns that should be updated if there is already a matching record in the database: DB::table('flights')->upsert([['departure' => 'Oakland', 'Destination' => 'San Diego', 'price' => 99], ['Departure' => 'Chicago', 'Destination' => 'New York', 'price' => 150]], ['Departure', 'Destination'], ['Price']); Note All databases except SQL Server require that the columns in the second argument of the upsert method have a primary or unique index. Automatic increment of IDs If the table has an automatic increment ID, use the insertGetId method to insert a record, and then get the ID: $id = DB::table('users')->insertGetId(['email' => '', 'votes' => 0]); Note When using PostgreSQL, the insertGetId method expects the column to automatically be incremented with the name id. If you want to retrieve the ID from another sequence, you can pass the column name as a second parameter to the insertGetId method. In addition to inserting records into the database, Query Builder can also update existing records using the update method. The update method, like the Insert method, accepts an array of pairs of columns and values that contain the columns to be updated. You can restrict the update query with the following places: $affected = DB::table('users') ->where('id', 1) ->update(['votes' => 1]); Sometimes you want to update or create an existing record in the database if there is no matching record. In this scenario, the updateOrInsert method can be used. The updateOrInsert method accepts two arguments: an array of conditions to find the record from, and Array of pairs of columns and values that contain the columns to be updated. The updateOrInsert method first attempts to find a matching database record using the columns and value pairs of the first argument. If the record exists, it is updated with the values in the second argument. If the record cannot be found, a new record is inserted with the merged attributes of both arguments: DB::table('users') ->updateOrInsert(['email' => '', 'name' 'name' 'John'], ['votes' => '2']); Updating JSON Columns When updating a JSON column, you should use the -> 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 practical methods for incrementing or decreting the value of a particular column. This is a link that provides a more expressive and concise interface than manually writing the update statement. Both methods accept at least one argument: the column to change. A second argument can optionally be passed to control the amount by which the column should be incremented or reduced: DB::table('users')->increment('votes'); DB::table('users')->increment('votes', 5); DB::table('users')->decrement('votes'); DB::table('users')->decrement('votes', 5); You can also specify additional columns to update during the operation: DB::table('users')->increment('votes', 1, ['name'

=> 'John']); Note Model events are not raised when the increment and decrement methods are used. Delete The Query Builder can also be used to delete records from the table using the Delete method. You can restrict delete statements by adding where clauses before calling the delete method: DB::table('users')->delete(); DB::table('users')->where('votes', '>', 100)->delete(); If you want to truncate the entire table, which removes all rows and resets the automatic increment ID to zero, you can use the gradation method: DB::table('users')->truncate(); Table Truncation & PostgreSQL When truncating a PostgreSQL database, the CASCADE behavior is applied. This means that all foreign key records in other tables are also deleted. Lock Pessimistic The Query Builder also includes some features that will help you do pessimistic locks on your selected statements. To execute the statement with a shared lock, you can use the sharedLock method for a query. A shared lock prevents the selected rows from being modified until your transaction is committed: DB::table('users')->where('votes', '>', 100)->sharedLock()->get(); Alternatively, you can use the lockForUpdate method. An update lock prevents the rows from being changed or selected with another shared lock: DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get(); Debug You can use the dd or while you create a query to overlay the query bindings and SQL bindings. The dd method displays the debugging information and then stops the request from executing. The dump method displays the debugging information, but still lets the request run: DB::table('users')->where('votes', '>', 100)->dd(); DB::table('users')->where('votes', '>', 100)->dump(); 100)->dump(); 100)->dump();