Model

A model is a PHP class that represents a database table and provides methods for interacting with the table's data.

Creating model by artisan command

```
php artisan make:model ModelName
```

This command will generate a new model file in the app/Models directory.

You can define the database table associated with the model and specify its columns, relationships, and other functionalities within the class.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class ModelName extends Model
```

```
{
    protected $table = 'table_name';
    protected $primaryKey = 'id';
    protected $fillable = ['column1', 'column2'];

    // Define relationships and other methods here
}
```

- The $table property specifies the name of the database table associated with the model.
- The $primaryKey property specifies the primary key column of the table (by default, it's assumed to be named "id").
- The $fillable property lists the columns that can be mass-assigned with values.

- Model different methods

```
// Retrieve all records
$models = ModelName::all();

// Retrieve a single record by primary key
    $model = ModelName::find($id);
```

```
// Querying records using conditions
$models = ModelName::where('column1', 'value')->get();
// Retrieve the first  record by first(); method.
$model=ModelName::first();
//delete a records by primary key
destroy($id);
```

Performing CRUD Operations with the help of model.

Laravel's Eloquent ORM provides a rich set of methods for performing CRUD (Create, Read, Update, Delete) operations on the associated database table. Here are some.

- Creating a new record.

```
$model = new ModelName;
$model->column1 = 'value1';
$model->column2 = 'value2';
$model->save();
```

- Retrieving Records

```
//retrieve all records
```

$test= Model_Name::all();

//retrieving a single record by primary key.

$test= Model_Name::find($id);

//Querying record using conditions

$test= Model_Name::where('column1','value')->get();

- Updating Record.

$model=Model_Name::find($id);

$model->column1='new value';

$model->save();

- Deleting the records

$model=Model_Name::find($id);

$model->delete();

Eloquent: Relationships

One-to-One Relationship

one-to-one relationship, a record in one table is associated with exactly one record in another table

example:

// User model

public function profile()

```
{
    return $this->hasOne(Profile::class);
}

// Profile model
public function user()
{
    return $this->belongsTo(User::class);
}
```

One-to-Many Relationship

In a one-to-many relationship, a record in one table can be associated with multiple records in another table. For example, a user may have multiple posts. To define a one-to-many relationship, you can use the hasMany and belongsTo methods. Here's an.

```
// User model
public function posts()
{
    return $this->hasMany(Post::class);
}

// Post model
public function user()
{
    return $this->belongsTo(User::class);
}
```

Many-to-Many Relationship:

In a many-to-many relationship, records in one table can be associated with multiple records in another table, and vice versa.

```
// User model
public function roles()
{
    return $this->belongsToMany(Role::class);
}

// Role model
public function users()
{
    return $this->belongsToMany(User::class);
}
```

DB Facades

```
use Illuminate\Support\Facades\DB;
```

Insert data in database use the DB Facades

```
DB::table('table_name')->insert([
```

‘column’=>’value’,

]);

Updating data in database use the DB Facades.

```
DB::table(‘table_name’)->where(‘column’, ‘value’)->update([

        'column1' => 'new_value1',
        'column2' => 'new_value2',
        ]);
```

Delete Data

```
DB::table('table_name')
    ->where('column', 'value')
    ->delete();
```

Selecting data

```
$result = DB::table('table_name')
    ->select('column1', 'column2')
    ->where('column', 'value')
    ->get();
```

Query Builder Methods:

The DB facade provides a fluent interface to chain multiple query builder methods for complex queries. You can use methods such as where, orWhere, orderBy, groupBy, join, leftJoin, limit, and more to construct your queries.

```
$result = DB::table('table_name')
    ->select('column1', 'column2')
    ->where('column', 'value')
    ->orWhere('column', 'value2')
    ->orderBy('column')
    ->limit(10)
    ->get();
```

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');
```

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;
}
```

Chunking Results

If you need to work with thousands of database records, consider using the chunk method provided by the DB facade. This method retrieves a small chunk of results at a time and feeds each chunk into a closure for processing. For example, let's retrieve the entire users table in chunks of 100 records at a time:

```
use Illuminate\Support\Collection;

use Illuminate\Support\Facades\DB;

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

    return false;
});
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. If you plan to update the retrieved 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 (Collection $users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

Aggregates

Count(), max(), min(), avg(), and sum()

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

```
use Illuminate\Support\Facades\DB;

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

$price = DB::table('orders')->max('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()) {
    // ...
}
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

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

The distinct method allows you to force the query to return distinct results. Use for return the unique value.

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

ADD Select Methods

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:

```
$data=DB::table(‘users’)->select(‘name’);

$users=$data->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.

```
$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();
```

Unions

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

```
use Illuminate\Support\Facades\DB;
```

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

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

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.

```
$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();
```

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();
```

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

Additional Where Clauses

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();
```

You may also provide a query object as the whereIn method's second argument

```
$activeUsers = DB::table('users')->select('id')->where('is_active', 1);

$users = DB::table('comments')
            ->whereIn('user_id', $activeUsers)
            ->get();
```

Sql query for above where in method.

```
select * from comments where user_id in (
    select id
    from users
    where is_active = 1
)
```

whereNull / whereNotNull / orWhereNull / orWhereNotNull

whereNull

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

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

whereNotNull

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

whereDate

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();
```

whereMonth

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();
```

whereDay

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();
```

whereYear

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();
```

whereTime

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

whereColumn

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();
```

Logical Grouping

Sometimes you may need to group several "where" clauses within parentheses in order to achieve your query's desired logical grouping. In fact, you should generally always group calls to the orWhere method in parentheses in order to avoid unexpected query

behavior. To accomplish this, you may pass a closure to the where method

```
$users = DB::table('users')
        ->where('name', '=', 'John')
        ->where(function (Builder $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')
```

Where Exists Clauses

The whereExists method allows you to write "where exists" SQL clauses. The whereExists method accepts a closure 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 (Builder $query) {
            $query->select(DB::raw(1))
                ->from('orders')
                ->whereColumn('orders.user_id', 'users.id');
        })
        ->get();
```

examples above will produce the following SQL

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

)

Full Text Where Clauses

The whereFullText and orWhereFullText methods may be used to add full text "where" clauses to a query for columns that have full text indexes. These methods will be transformed into the appropriate SQL for the underlying database system by Laravel. For example, a MATCH AGAINST clause will be generated for applications utilizing MySQL

```
$users = DB::table('users')
        ->whereFullText('bio', 'web developer')
        ->get();
```

Ordering, Grouping, Limit & Offset

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();
```

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();
```

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();
```

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]);
```

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']
    );
```

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();
```

Debugging

You may use the dd and dump methods while building a query to dump the current 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 continue executing

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

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

Pessimistic Locking

The query builder also includes a few functions to help you achieve "pessimistic locking" when

executing your select statements. To execute a statement with a "shared lock", you may call the sharedLock method. 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 may use the lockForUpdate method. A "for update" lock prevents the selected records from being modified or from being selected with another shared lock

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

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