

Meet Eloquent ORM

- Eloquent is an object-relational mapper (ORM)
- Eloquent models represent database tables.
- Models can be used to perform operations on data
- Eloquent also supports relationships between models
- It provides layer between application and database

Naming Convention

- Table name: brands Model name: Brand
- Table name: product_details Model name: ProductDetails
- If we follow this convention no need to define table name inside model. Other wise we need to define.

```
class Flight extends Model
{
    // ...
}
```

```
class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Naming Convention

- Model assume that corresponding database table has a primary key column named id.
- Eloquent assumes that the primary key is an incrementing integer value
- If necessary, you may define a protected \$primaryKey property

```
class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

```
public $incrementing = false;
```

```
protected $keyType = 'string';
```

Naming Convention

Model expects `created_at` and `updated_at` columns exist on corresponding database table.

Eloquent will automatically set these column's values when models are created or updated.

If you do not want timestamp automatically managed by Eloquent, you should define a `$timestamps` property false.

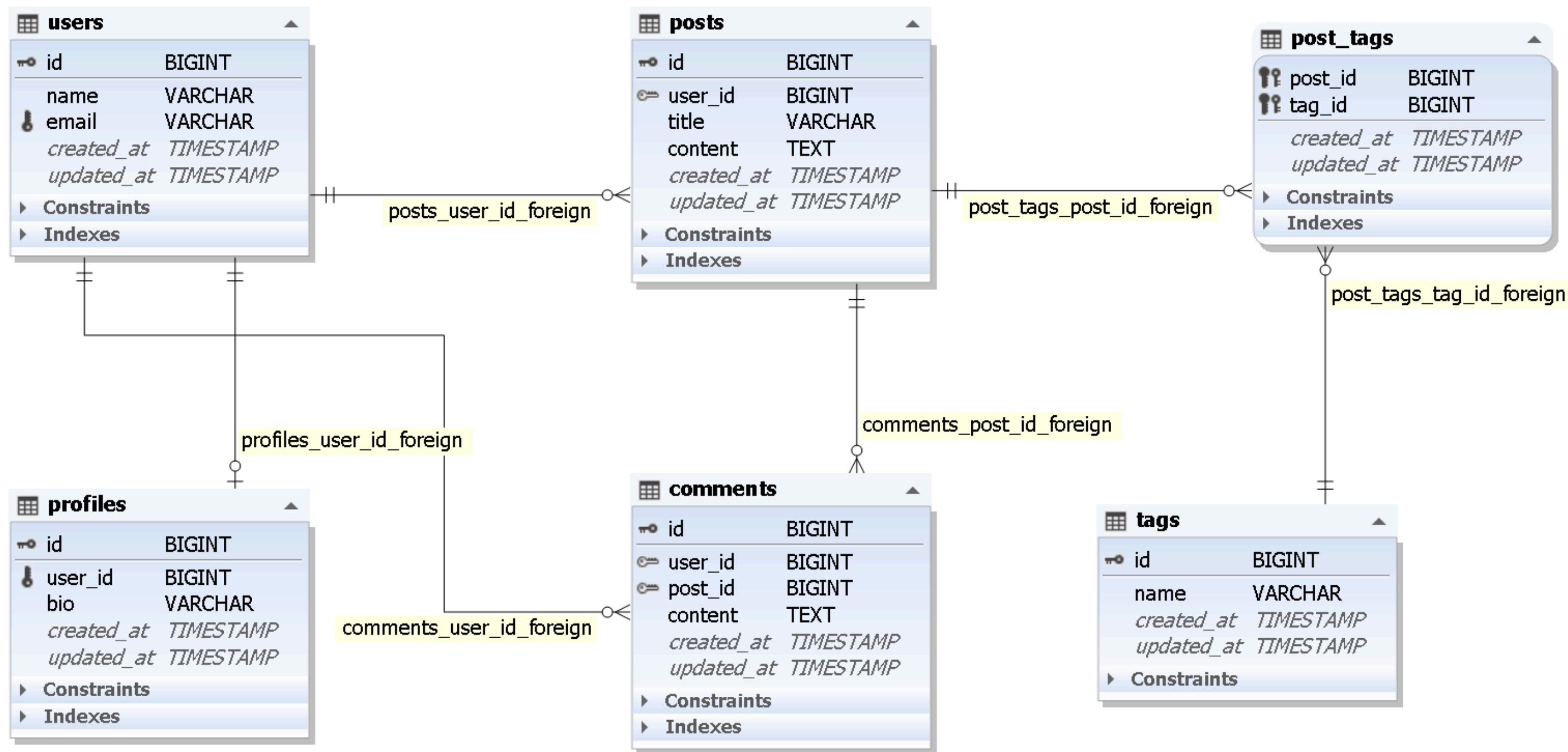
```
class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

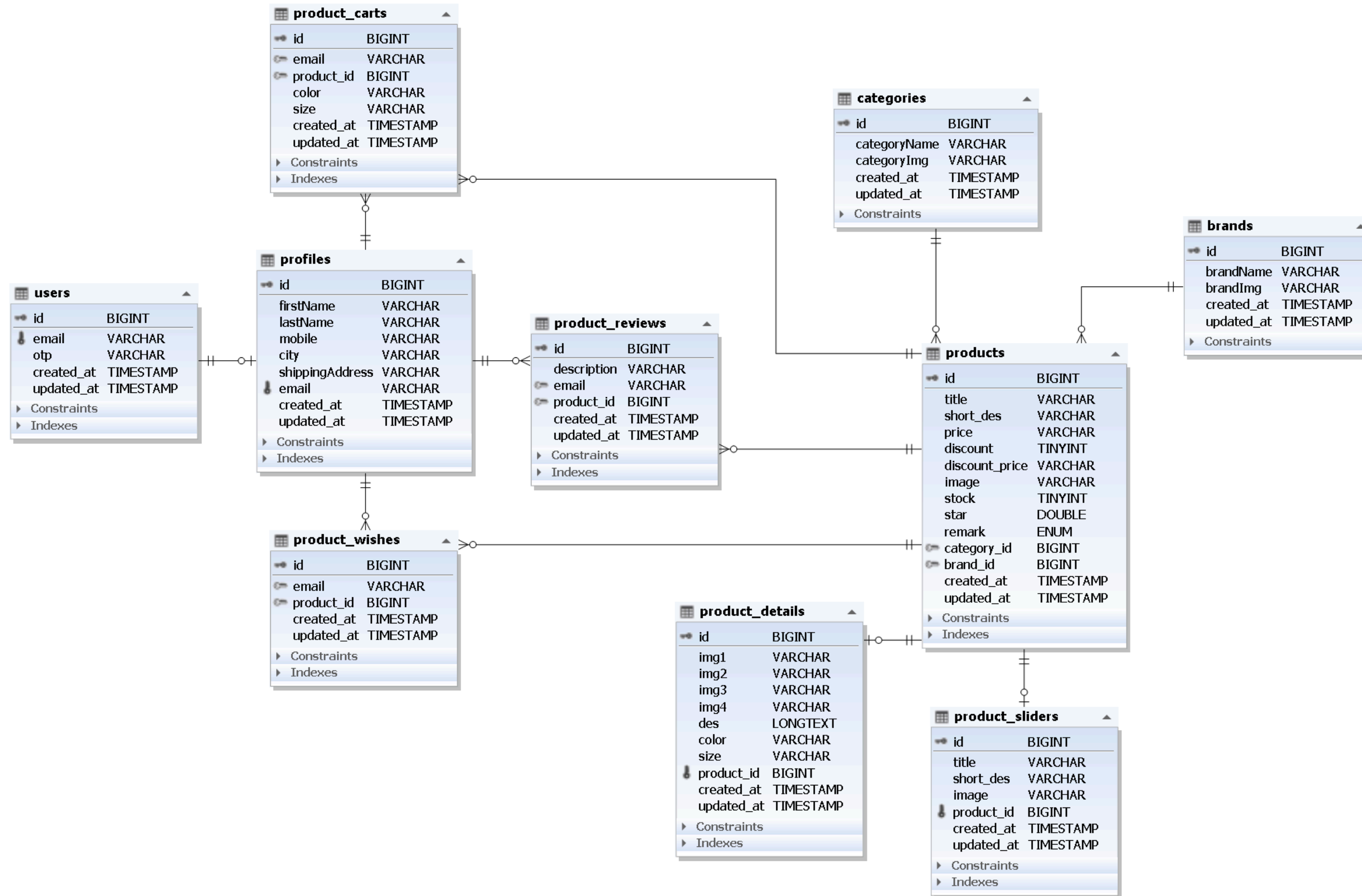
Naming Convention

By default model instance will not contain any attribute values.

But you can define an \$attributes property on your model as you want.

```
class Brand extends Model
{
    protected $attributes = [
        'brandName' => 'default name',
        'brandImg' => 'default img',
    ];
}
```





Insert Using Model

The **create** method to "save" a new model using a single PHP statement.

In Laravel, the **fillable** property is used to define which fields in a model can be mass assigned.

```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::create($request->input());
    }
}
```

```
class Brand extends Model
{
    protected $fillable = ['brandName', 'brandImg'];
}
```


Update Using Model

The **update** method expects an array of column and value pairs representing the columns that should be updated

In Laravel, the **fillable** property is used to define which fields in a model can be mass assigned.

```
public function DemoAction(Request $request)
{
    return Brand::where('id',$request->id)
        ->update($request->input());
}
```

```
class Brand extends Model
{
    protected $fillable = ['brandName', 'brandImg'];
}
```

Why we write fillable inside model

There are a few reasons why you might want to use the fillable property in Laravel

- **Security:** It helps protect your application from mass assignment attacks.
- **Performance:** It can improve the performance of your application by reducing the number of fields that need to be checked when mass assigning data.
- **Readability:** It makes your code more readable by explicitly defining which fields can be mass assigned.
- **Consistency:** It ensures that all of your models have the same set of fillable fields, which can make your code more consistent.
- **Extensibility:** It allows you to easily add or remove fillable fields as your application evolves.

Update or Create Using Model

The **updateOrCreate()** method in Laravel is used to update an existing record in the database if it exists, or create a new record if it doesn't exist.

```
public function DemoAction(Request $request)
{
    return Brand::updateOrCreate(
        ['brandName' => $request->brandName],
        $request->input()
    );
}
```

Delete Using Model

The **delete()** method in Laravel's Eloquent ORM is used to delete a record from the database.

```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::where('id',$request->id)
            ->delete();
    }
}
```

Increment & Decrement

Laravel provides convenient methods for [incrementing](#) or [decrementing](#) the value of a given column.



```
public function DemoAction()
{
    Product::where('id', '=', 1)
        ->increment('price', 1000);
    //->decrement('price', 1000);
    //->increment('price');
    //->decrement('price');
}
```

Retrieving All Rows

- Use the **get()** method to execute
- Use the **all()** method to execute

```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::get();
    }
}
```

```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::all();
    }
}
```


Retrieving Single Row



```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::first();
    }
}
```



```
class DemoController extends Controller
{
    public function DemoAction(Request $request)
    {
        return Brand::find(1);
    }
}
```


Retrieving List Of Column Values



```
public function DemoAction(Request $request)
{
    return Product::pluck('price');
}
```



```
public function DemoAction(Request $request)
{
    return Product::pluck('price', 'name');
}
```

Aggregates

- Methods for retrieving aggregate values like `count`, `max`, `min`, `avg`, and `sum`.
- Call any of these methods after constructing query



```
Product::count('price');  
Product::avg('price');  
Product::sum('price');  
Product::max('price');  
Product::min('price');
```

Select Clause

- The **select()** method allows you to specify the columns
- To return distinct results use the **distinct()** method



```
public function DemoAction()
{
    Product::select('title', 'price')->get();
}
```



```
public function DemoAction()
{
    Product::select('title')->distinct()->get();
}
```

Basic Where Clauses

The `where()` method allows you to filter the results.

- `=` (equal to)
- `!=` (not equal to)
- `<` (less than)
- `<=` (less than or equal to)
- `>` (greater than)
- `>=` (greater than or equal to)
- `LIKE` (contains)
- `NOT LIKE` (does not contain)
- `IN` (is in the list)
- `NOT IN` (is not in the list)

Advance Where Clauses

- The `orWhere` method to join a clause to the query using the or operator.
- The `whereNot` and `orWhereNot` methods may be used to negate a given group of query constraints.
- The `whereBetween` method verifies that a column's value is between two values .
- The `whereNotBetween` method verifies that a column's value lies outside of two values.
- The `whereBetweenColumns` method verifies that a column's value is between the two values of two columns in the same table row.
- The `whereNotBetweenColumns` method verifies that a column's value lies outside the two values of two columns in the same table row.
- The `whereIn` method verifies that a given column's value is contained within the given array.
- The `whereNotIn` method verifies that the given column's value is not contained in the given array.
- The `whereNull` method verifies that the value of the given column is NULL.
- he `whereNotNull` method verifies that the column's value is not NULL.
- The `whereDate` method may be used to compare a column's value against a date.
- The `whereMonth` method may be used to compare a column's value against a specific month.
- The `whereDay` method may be used to compare a column's value against a specific day of the month.
- The `whereYear` method may be used to compare a column's value against a specific year.
- The `whereTime` method may be used to compare a column's value against a specific time.
- The `whereColumn` method may be used to verify that two columns are equal.

Ordering, Grouping, Limit

- The `orderBy` method allows you to sort the results of the query by a given column
- The `latest` and `oldest` methods allow you to easily order results by date
- The `inRandomOrder` method may be used to sort the query results randomly
- The `groupBy` and `having` methods may be used to group the query results
- 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

Paginate

Display simple "Next" and "Previous" links in your application's UI, use the `simplePaginate` method to perform a single, efficient query

```
class DemoController extends Controller
{
    public function DemoAction()
    {
        return Product::simplePaginate(2);
    }
}
```


Paginate

The `paginate` method counts the total number of records matched by the query before retrieving the records from the database.

```
class DemoController extends Controller
{
    public function DemoAction()
    {
        return Product::paginate(2);
    }
}
```

```
public function DemoAction()
{
    return Product::paginate(
        $perPage = 5,
        $columns = ['*'],
        $pageName = 'pageName'
    );
}
```

Understanding Eloquent Relationship Term

Has Relationship:

- It indicates that one model **"has"** one or more related models.
- The model that defines the relationship has the foreign key column. This means that the model that defines the relationship is the **"parent"** model, and the model that is related to it is the **"child"** model.

Belongs Relationship:

- The **"belongs"** relationship, also known as the **"inverse"** relationship, defines the inverse of a **"has"** relationship
- This means that the model that is related to the other model is the **"parent"** model, and the model that defines the relationship is the **"child"** model

Understanding Eloquent Relationship Term

Relationship	Model with foreign key	Model that defines the relationship
hasOne	Child	Parent
belongsTo	Parent	Child

One to One & inverse Relationship

The **hasOne()** method in Laravel is used to define a **one-to-one** relationship between two models.

The **belongsTo()** method in Laravel is used to define a **one-to-one inverse** relationship between two models.

```
class Profile extends Model
{
    function user():BelongsTo{
        return $this->belongsTo(User::class);
    }
}
```

```
class User extends Model
{
    function profile():HasOne{
        return $this->hasOne(Profile::class);
    }
}
```

```
User::with('profile')->get();
```

One to Many & inverse Relationship

The **hasMany()** method in Laravel is used to define a **one-to-many** relationship between two models.

The **belongsTo()** method in Laravel is used to define a **one-to-many inverse** relationship between two models.



```
class User extends Model
{
    function post():HasMany{
        return $this->hasMany(Post::class);
    }
}
```



```
class Post extends Model
{
    function user(){
        return $this->belongsTo(User::class);
    }
}
```



```
User::with('post')->get();
```

Many to Many Relationship

The **belongsToMany()** method in Laravel is used to define a **many-to-many** relationship between models.



```
class Post extends Model
{
    function tags():BelongsToMany{
        return $this->belongsToMany(Tag::class,'post_tags');
    }
}
```



```
class Tag extends Model
{
    function posts():BelongsToMany{
        return $this->belongsToMany(Post::class,'post_tags');
    }
}
```

Has One Of Many

latestOfMany and **oldestOfMany** methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable.

```
class User extends Model
{
    function post():HasOne{
        return $this->hasOne(Post::class)->latestOfMany();
    }
}
```

```
class User extends Model
{
    function post():HasOne{
        return $this->hasOne(Post::class)->oldestOfMany();
    }
}
```