Assignment Of Module 17

Syed Amir hamza

1. Explain what Laravel's query builder is and how it provides a simple and elegant way to interact with databases.

Laravel's query builder is a feature that provides a convenient and expressive way to interact with databases in Laravel, a popular PHP framework. It allows developers to build database queries using a fluent, chainable interface, instead of writing raw SQL statements.

The query builder provides a set of methods that represent different parts of a SQL query, such as selecting columns, applying conditions, joining tables, and ordering results. These methods can be chained together to construct complex queries in a readable and intuitive manner.
Here are some key benefits of using Laravel's query builder:

***Simplified syntax**: The query builder offers a more concise and readable syntax compared to writing raw SQL queries. It uses method calls and method chaining to build queries, making it easier to understand and maintain the code.*

***Database agnostic**: Laravel's query builder is designed to work with multiple database systems, including MySQL, PostgreSQL, SQLite, and SQL Server. It provides a consistent API across different database engines, allowing developers to write portable database queries.*

***Parameter binding**: The query builder automatically handles parameter binding, which helps prevent SQL injection attacks. Instead of concatenating values directly into the query, you can use placeholders, and the query builder will bind the values securely.*

***ORM integration**: Laravel's query builder seamlessly integrates with its Object-Relational Mapping (ORM) feature called Eloquent. You can use the query builder to build queries for Eloquent models, combining the power of both approaches. This allows you to easily switch between raw queries and working with models.*

***Advanced query building**: The query builder supports a wide range of query building capabilities, such as subqueries, unions, joins, aggregate functions, and more. It provides a rich set of methods for constructing complex queries without needing to resort to raw SQL.*

Overall, Laravel's query builder offers a clean and intuitive way to interact with databases in PHP applications. It abstracts away the complexities of writing raw SQL queries and provides a consistent, database-agnostic API for building and executing queries. This results in more maintainable and readable code, as well as enhanced security through parameter binding.

2.Write the code to retrieve the "excerpt" and "description" columns from the "posts" table using Laravel's query builder. Store the result in the $posts variable. Print the $posts variable.

```php
<?php

use Illuminate\Support\Facades\DB;

$posts = DB::table('posts')
    ->select('excerpt', 'description')
    ->get();

print_r($posts);

?>
```

*In the above code, we first import the **DB** facade from the **Illuminate\Support\Facades** namespace. This allows us to use Laravel's query builder methods.*

*Next, we use the **table()** method on the **DB** facade to specify the table we want to query, which in this case is "posts". Then, we chain the **select()** method to specify the columns we want to retrieve, which are "excerpt" and "description".*

*Finally, we call the **get()** method to execute the query and retrieve the results. The results are stored in the **$posts** variable. We then use **print_r()** to print the contents of the **$posts** variable.*
Note that you'll need to have Laravel properly set up and the necessary database connection configured for this code to work correctly.

3.Describe the purpose of the distinct() method in Laravel's query builder. How is it used in conjunction with the select() method?

Distinct: The distinct method ensures that only distinct (unique) values are returned for the specified columns.

In Laravel, The **distinct()** method in Laravel's query builder is used to retrieve unique values from a specific column or a combination of columns in the result set. It ensures that duplicate values are removed, and only distinct values are returned.
When used in conjunction with the **select()** method, the **distinct()** method modifies the query to include the **DISTINCT** keyword in the underlying SQL query. This instructs the database to return only unique values for the selected columns.
Here's an example to illustrate the usage of **distinct()** with **select()** in Laravel's query builder:

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

$uniqueNames = DB::table('users')
    ->select('name')
    ->distinct()
    ->get();

foreach ($uniqueNames as $name) {
    echo $name->name . "\n";
}
```

In the above code, we're querying the "users" table and selecting the "name" column. By adding the **distinct()** method after the **select()** method, we ensure that only unique names are retrieved.
The resulting SQL query would be something like: **SELECT DISTINCT name FROM users**.
The retrieved rows are then looped through in the **foreach** loop, and the **name** property of each row is echoed. This will output the unique names from the "users" table.

It's important to note that **distinct()** affects the entire query, not just the columns specified in the **select()** method. So if you have additional conditions or joins in your query, the **distinct()** method will apply to all the selected columns and their combinations.

By using **distinct()** in conjunction with **select()**, you can effectively filter out duplicate values and retrieve only distinct values from your database queries.

4.Write the code to retrieve the first record from the "posts" table where the "id" is 2 using Laravel's query builder. Store the result in the $posts variable. Print the "description" column of the $posts variable.

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

$posts = DB::table('posts')
    ->where('id', 2)
    ->first();

if ($posts) {
    echo $posts->description;
} else {
    echo "No post found.";
}
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*
*Next, we chain the **where()** method to add a condition to retrieve the record where the "id" column is equal to 2.*
*We then call the **first()** method to retrieve the first matching record from the table based on the condition.*
*If a matching record is found, we print the "description" column of the **$posts** variable using **$posts->description**. If no record is found, we print "No post found."*
Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

5.Write the code to retrieve the "description" column from the "posts" table where the "id" is 2 using Laravel's query builder. Store the result in the $posts variable. Print the $posts variable.

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

$posts = DB::table('posts')
    ->where('id', 2)
    ->pluck('description');

print_r($posts);
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*
*Next, we chain the **where()** method to add a condition to retrieve the record where the "id" column is equal to 2.*
*We then use the **pluck()** method to retrieve only the value of the "description" column from the resulting record.*
*The **pluck()** method returns a single value, so we directly store the result in the **$posts** variable.*
*Finally, we use **print_r()** to print the contents of the **$posts** variable.*
Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

6. Explain the difference between the first() and find() methods in Laravel's query builder. How are they used to retrieve single records?

*In Laravel's query builder, the **first()** and **find()** methods are used to retrieve single records from the database, but they have slight differences in how they work:*

***1. first() method**: The **first()** method is used to retrieve the first record that matches the query conditions. It returns a single object representing the first matching row. It is commonly used when you want to retrieve a single record based on specific conditions or to get the first record in a table.*
Example usage:

```
$post = DB::table('posts')
    ->where('category', 'news')
    ->orderBy('created_at')
    ->first();
```

*In the above example, **first()** retrieves the first record from the "posts" table that has a category of "news" and orders the results by the **created_at** column.*

***2.find() method**: The **find()** method is used to retrieve a record by its primary key value. It specifically targets a record based on its primary key column. It returns a single object representing the matching row, or **null** if no matching record is found.*
Example usage:

```
$post = DB::table('posts')->find(1);
```

*1.In the above example, **find(1)** retrieves the record from the "posts" table where the primary key (usually named **id**) is 1. It directly retrieves the record without needing to specify any additional conditions.*

*To summarize, the main difference between **first()** and **find()** is that **first()** is used when you want to retrieve the first record based on specified conditions, whereas **find()** is used when you want to retrieve a record by its primary key value.*

*Both methods return a single record, but **first()** allows you to specify conditions beyond the primary key column, while **find()** targets the record directly based on the primary key.*

7.Write the code to retrieve the "title" column from the "posts" table using Laravel's query builder. Store the result in the $posts variable. Print the $posts variable.

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

$posts = DB::table('posts')
    ->select('title')
    ->get();

print_r($posts);
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*
*Next, we chain the **select()** method to specify that we want to retrieve the "title" column from the table.*
*We then call the **get()** method to execute the query and retrieve the results. The results are stored in the **$posts** variable.*
*Finally, we use **print_r()** to print the contents of the **$posts** variable.*
Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

8.Write the code to insert a new record into the "posts" table using Laravel's query builder. Set the "title" and "slug" columns to 'X', and the "excerpt" and "description" columns to 'excerpt' and 'description', respectively. Set the "is_published" column to true and the "min_to_read" column to 2. Print the result of the insert operation.

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


$result = DB::table('posts')->insert([
    'title' => 'X',
    'slug' => 'X',
    'excerpt' => 'excerpt',
    'description' => 'description',
    'is_published' => true,
    'min_to_read' => 2
]);

print_r($result);
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*

*We then call the **insert()** method and pass an associative array as its argument. The keys of the array represent the column names, and the corresponding values are the data we want to insert into the columns.*

In this example, we set the "title" and "slug" columns to 'X', the "excerpt" column to 'excerpt', the "description" column to 'description', the "is_published" column to true, and the "min_to_read" column to 2.

*The **insert()** method returns a boolean indicating the success of the insert operation. We use **print_r()** to print the result of the insert operation.*

Need to be sure Laravel properly set up and a valid connection to the database configured for this code to work correctly.

9.Write the code to update the "excerpt" and "description" columns of the record with the "id" of 2 in the "posts" table using Laravel's query builder. Set the new values to 'Laravel 10'. Print the number of affected rows.

```php
$affectedRows = DB::table('posts')
    ->where('id', 2)
    ->update([
        'excerpt' => 'Laravel 10',
        'description' => 'Laravel 10'
    ]);

echo "Number of affected rows: " .
$affectedRows;
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*

*Next, we chain the **where()** method to add a condition to target the record with the "id" of 2. We then call the **update()** method and pass an associative array as its argument. The keys of the array represent the column names that we want to update, and the corresponding values are the new values we want to set.*

In this example, we set the "excerpt" and "description" columns to 'Laravel 10'.
*The **update()** method returns the number of affected rows, indicating how many records were updated. We echo the number of affected rows using **echo** statement.*

Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

10.Write the code to delete the record with the "id" of 3 from the "posts" table using Laravel's query builder. Print the number of affected rows.

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

$affectedRows = DB::table('posts')
    ->where('id', 3)
    ->delete();

echo "Number of affected rows: " .
$affectedRows;
;
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*

*Next, we chain the **where()** method to add a condition to target the record with the "id" of 3. We then call the **delete()** method to delete the record that matches the condition.*

*The **delete()** method returns the number of affected rows, indicating how many records were deleted. We echo the number of affected rows using **echo** statement.*

Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

11. Explain the purpose and usage of the aggregate methods count(), sum(), avg(), max(), and min() in Laravel's query builder. Provide an example of each.

*In Laravel's query builder, the aggregate methods **count()**, **sum()**, **avg()**, **max()**, and **min()** are used to perform calculations on a column or a set of columns in a database table. These methods allow you to retrieve aggregate values such as the count of rows, the sum of values, the average of values, the maximum value, and the minimum value. Here's an explanation and example of each aggregate method:*

***1.count() method**: The **count()** method is used to retrieve the number of rows that match a specific condition or the total number of rows in a table.*
Example:

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

*In this example, **count()** retrieves the total number of rows in the "users" table.*

***2. sum() method**: The **sum()** method is used to calculate the sum of values in a column.*
Example:

```
$total = DB::table('orders')->sum('amount');
```

*In this example, **sum('amount')** calculates the sum of the "amount" column in the "orders" table.*

***3.avg() method**: The **avg()** method is used to calculate the average (mean) of values in a column.*
Example:

```
$average = DB::table('products')->avg('price');
```

*In this example, **avg('price')** calculates the average of the "price" column in the "products" table.*

***4. max() method**: The **max()** method is used to retrieve the maximum value from a column.*
Example:

```
$maxValue = DB::table('products')->max('price');
```

*In this example, **max('price')** retrieves the maximum value from the "price" column in the "products" table.*

***5. min() method**: The **min()** method is used to retrieve the minimum value from a column.*
***Example**:*

```
$minValue = DB::table('products')->min('price');
```

In this example, min('price') retrieves the minimum value from the "price" column in the "products" table.

12. Describe how the whereNot() method is used in Laravel's query builder. Provide an example of its usage.

*In Laravel's query builder, the **whereNot()** method is used to add a "where not" condition to a query. It allows you to retrieve records that do not match a specific condition. The **whereNot()** method is often used in combination with other query builder methods to create more complex and specific conditions.*
*Here's an example to illustrate the usage of the **whereNot()** method:*

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


$users = DB::table('users')
    ->whereNot('status', 'active')
    ->get();
```

*In the above example, we use the **DB** facade to access Laravel's query builder methods. We specify the "users" table using the **table()** method.*

*We then chain the **whereNot()** method to add a condition. In this case, the condition is that the "status" column should not be equal to 'active'. This means we want to retrieve users with a status other than 'active'.*

*The **whereNot()** method takes two arguments: the column name and the value to compare against. It adds a "where not" condition to the query, which excludes records that match the condition.*

*Finally, we call the **get()** method to execute the query and retrieve the results. The results are stored in the **$users** variable.*

*This example would retrieve all the users from the "users" table whose status is not 'active'. The **whereNot()** method provides a convenient way to negate a specific condition in a query, allowing you to filter and retrieve records that do not meet certain criteria.*

13. Explain the difference between the exists() and doesntExist() methods in Laravel's query builder. How are they used to check the existence of records?

In Laravel's query builder, the **exists()** and **doesntExist()** methods are used to check the existence of records in a table. However, they differ in terms of the condition they evaluate and the way they are used.

***1.exists() method**: The **exists()** method is used to check if any records exist in the result set of a query. It returns **true** if the query returns at least one record and **false** otherwise.*
Example:

```
$exists = DB::table('users')->where('status',
'active')->exists();
```

*In this example, **exists()** checks if there are any records in the "users" table where the status is 'active'. If at least one such record exists, the method will return **true**.*

***2.doesntExist() method**: The **doesntExist()** method is used to check if no records exist in the result set of a query. It returns **true** if the query returns no records and **false** if there is at least one record.*
Example:

```
$doesntExist = DB::table('users')
        ->where('status', 'active')
        ->doesntExist();
```

*In this example, **doesntExist()** checks if there are no records in the "users" table where the status is 'active'. If no such record exists, the method will return **true**.*

*To summarize, the **exists()** method is used to check if any records exist that meet a specific condition, while the **doesntExist()** method is used to check if no records exist that meet a specific condition.*

These methods are useful when you need to verify the presence or absence of data before performing further operations or making decisions based on the existence of records in the database.

14. Write the code to retrieve records from the "posts" table where the "min_to_read" column is between 1 and 5 using Laravel's query builder. Store the result in the $posts variable. Print the $posts variable.

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


$posts = DB::table('posts')
    ->whereBetween('min_to_read', [1, 5])
    ->get();

print_r($posts);
```

*In the above code, we use the **DB** facade to access Laravel's query builder methods. We specify the "posts" table using the **table()** method.*
*We then chain the **whereBetween()** method to add a condition. In this case, we specify that the "min_to_read" column should be between 1 and 5. This means we want to retrieve records where the "min_to_read" value falls within that range.*
*The **whereBetween()** method takes two arguments: the column name and an array containing the minimum and maximum values for the range.*
*Finally, we call the **get()** method to execute the query and retrieve the results. The results are stored in the **$posts** variable.*
This example would retrieve all the records from the "posts" table where the "min_to_read" column value is between 1 and 5.
Make sure you have Laravel properly set up and a valid connection to the database configured for this code to work correctly.

15.Write the code to increment the "min_to_read" column value of the record with the "id" of 3 in the "posts" table by 1 using Laravel's query builder. Print the number of affected rows.

```php
$posts = DB::table('posts')
    ->select('excerpt', 'description')
    ->get();

print_r($posts);
```

*In the above code, we first import the **DB** facade from the **Illuminate\Support\Facades** namespace. This allows us to use Laravel's query builder methods.*

*Next, we use the **table()** method on the **DB** facade to specify the table we want to query, which in this case is "posts". Then, we chain the **select()** method to specify the columns we want to retrieve, which are "excerpt" and "description".*

*Finally, we call the **get()** method to execute the query and retrieve the results. The results are stored in the **$posts** variable. We then use **print_r()** to print the contents of the **$posts** variable.*

Note that you'll need to have Laravel properly set up and the necessary database connection configured for this code to work correctly.