

Module 17 Assignment
Submitted by Md. Ratul Hossain
Date: 12/06/2023

Answer to question no 1:

Laravel's query builder is a powerful feature of the Laravel framework that provides a convenient and expressive way to interact with databases. It allows developers to build database queries using a fluent and intuitive interface, rather than writing raw SQL statements.

With the query builder, you can perform database operations such as retrieving records, inserting, updating, and deleting data. It supports various database systems, including MySQL, PostgreSQL, SQLite, and SQL Server.

Here are some key features and benefits of Laravel's query builder:

1. **Fluent interface:** The query builder uses a fluent interface, which means you can chain methods together to construct your queries. This results in clean and readable code that closely resembles natural language.
2. **Parameter binding:** The query builder handles parameter binding automatically, which helps prevent SQL injection attacks. It securely binds the input values to the query, ensuring that user-supplied data is properly escaped.
3. **Select statements:** You can specify the columns you want to retrieve using the ``select`` method. You can also perform aggregate functions, apply conditions, join tables, and order the results with ease.
4. **Insert statements:** Inserting data into the database is straightforward with the query builder. You can use the ``insert`` method to insert a single row or the ``insert`` method with an array of rows to insert multiple rows at once.

5. Update statements: Updating records is done using the ``update`` method, which allows you to set the column values to new values based on specified conditions.

6. Delete statements: The query builder provides the ``delete`` method to remove records from the database. You can apply conditions to delete specific rows or remove all rows from a table.

7. Eloquent integration: Laravel's query builder seamlessly integrates with its ORM (Object-Relational Mapping) called Eloquent. Eloquent allows you to define models that represent database tables and easily perform queries using the query builder methods.

8. Database agnostic: The query builder supports multiple database systems, allowing you to write database-agnostic code. You can switch between different database systems without having to modify your queries.

Overall, Laravel's query builder simplifies the process of interacting with databases by providing a clean and intuitive syntax. It abstracts away the complexities of raw SQL and promotes best practices, making database operations more manageable and less error-prone.

Answer to question no 2:

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

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

print_r($posts);
```

Answer to question no 3:

The `distinct()` method in Laravel's query builder is used to retrieve unique records from a database table. It ensures that the query only returns distinct (unique) values for the specified columns.

When used in conjunction with the `select()` method, the `distinct()` method modifies the SELECT statement to include the DISTINCT keyword, indicating that only unique records should be retrieved based on the specified columns.

Here's an example to illustrate its usage:


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

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

In this example, i retrieving unique names from the "users" table. The ``select()`` method specifies that i only want to retrieve the "name" column. By chaining the ``distinct()`` method, i ensure that only distinct names are returned in the result.

It's important to note that the ``distinct()`` method affects the entire query and not just a specific column. So if has multiple columns in the ``select()`` method, the distinctness will apply to all the columns together, not individually.

Additionally, i can also pass specific columns to the ``distinct()`` method to apply the distinctness only to those columns, like this:



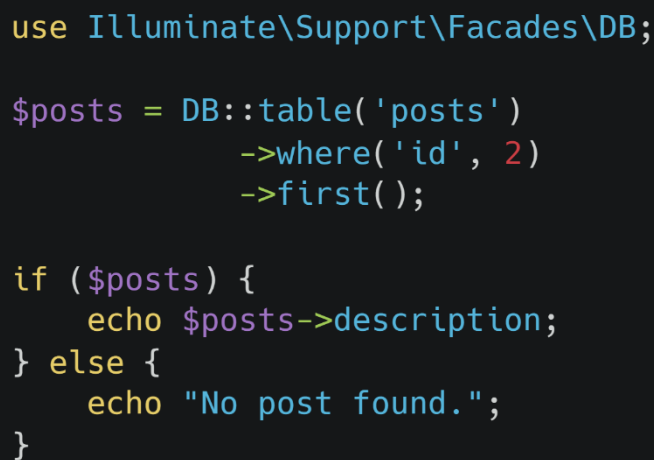
```
$uniqueNames = DB::table('users')  
  ->select('name', 'email')  
  ->distinct('name')  
  ->get();
```

In this case, the distinctness is applied only to the "name" column, and the "email" column is not considered when determining uniqueness.

The ``distinct()`` method is useful when i want to retrieve a list of unique values from a column or a combination of columns in my database table. It helps eliminate duplicate records and allows me to work with unique data in my queries.

Answer to question no 4:

Here's an example code snippet that retrieves the first record from the "posts" table where the "id" is 2 using Laravel's query builder, stores the result in the `\$posts` variable, and prints the "description" column of the `\$posts` variable:



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

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

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

In this code, i use the `table` method to specify the "posts" table. The `where` method is used to add a condition that checks for the "id" column to be equal to 2.

The `first` method retrieves the first record that matches the condition. If a matching record is found, it is stored in the `\$posts` variable. Then i check if `\$posts` is not null before attempting to access its properties.

Finally, i use `echo` to print the "description" column of the `\$posts` variable. If no post is found, i display a "No post found." message.

Answer to question no 5:

Here's a code snippet that retrieves the "description" column from the "posts" table where the "id" is 2 using Laravel's query builder, stores the result in the `\$posts` variable, and prints the `\$posts` variable:



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

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

print_r($posts);
```

In this code, i use the `table` method to specify the "posts" table. The `where` method adds a condition that checks for the "id" column to be equal to 2.


The `value` method retrieves the value of the "description" column from the matching record. It directly returns the single value without wrapping it in an array or object.

Finally, i use `print_r` to print the `\$posts` variable, which will display the value of the "description" column.

Answer to question no 6:

In Laravel's query builder, the `first()` and `find()` methods are used to retrieve single records from a database table. However, they have slight differences in their behavior and how they are used.

1. `first()`: This method is used to retrieve the first record that matches the query conditions. It returns a single instance of the model or `null` if no matching record is found. The `first()` method does not require any arguments and is typically used with query constraints such as `where()` or `orderBy()` to specify the conditions for retrieving the first record. Here's an example:



```
$user = DB::table('users')
    ->where('age', '>', 18)
    ->orderBy('created_at')
    ->first();
```

In this example, the `first()` method is used to retrieve the first user who is older than 18 years and has the oldest creation date.

2. `find()`: This method is used to retrieve a record by its primary key. It accepts the primary key value as an argument and returns the corresponding model instance or `null` if the record is not found. The `find()` method is commonly used when you know the primary key of the record you want to retrieve. Here's an example:



```
$user = User::find(1);
```

In this example, the `find()` method is used to retrieve the user record with the primary key value of 1.

To summarize, the `first()` method is used to retrieve the first record matching the specified query conditions, while the `find()` method is used to retrieve a record by its primary key. Both methods return a single record, but `first()` is more flexible as it allows you to specify various query constraints, whereas `find()` is primarily used when you already know the primary key of the record you want to retrieve.

Answer to question no 7:

Here's the code to retrieve the "title" column from the "posts" table using Laravel's query builder:



```
$posts = DB::table('posts')->pluck('title');  
print_r($posts);
```

In this example, the `pluck()` method is used to retrieve only the "title" column from the "posts" table. The result is stored in the `$posts` variable. Finally, the `print_r()` function is used to print the contents of the `$posts` variable.

Answer to question no 8:

Here's the code to insert a new record into the "posts" table using Laravel's query builder with the specified column values:

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

In this example, the `insert()` method is used to insert a new record into the "posts" table. The column names and their corresponding values are provided as an associative array within the `insert()` method.

After executing the insert operation, the result of the operation is stored in the `$result` variable. Finally, the `print_r()` function is used to print the result of the insert operation. The result will be `true` if the insertion was successful, or `false` otherwise.

Answer to question no 9:

Here's 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:

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

print_r($affectedRows);
```

In this example, the `update()` method is used to update the specified columns of the record with the "id" of 2 in the "posts" table. The `where()` method is used to specify the condition for selecting the record with the desired "id". The `update()` method accepts an associative array where the column names and their new values are provided.

After executing the update operation, the number of affected rows is stored in the `$affectedRows` variable. Finally, the `print_r()` function is used to print the number of affected rows.

Answer to question no 10:

Here's the code to delete the record with the "id" of 3 from the "posts" table using Laravel's query builder:



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

print_r($affectedRows);
```

In this example, the `delete()` method is used to delete the record with the specified "id" from the "posts" table. The `where()` method is used to specify the condition for selecting the record with the desired "id".


After executing the delete operation, the number of affected rows is stored in the `$affectedRows` variable. Finally, the `print_r()` function is used to print the number of affected rows.

Answer to question no 11:

In Laravel's query builder, the aggregate methods ``count()``, ``sum()``, ``avg()``, ``max()``, and ``min()`` are used to perform calculations on database columns and retrieve aggregate results from a query. These methods allow you to obtain useful information about your data without retrieving all the individual rows.

1. ``count()``:

The ``count()`` method is used to retrieve the total number of records matching a specific query. It returns an integer value representing the count. Here's an example:



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

This query will return the total count of records in the ``users`` table.

2. ``sum()``:

The ``sum()`` method calculates the sum of a specific column's values. It is typically used with numeric columns. Here's an example:



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

This query will calculate the sum of the `amount` column in the `orders` table, giving you the total sales.

3. `avg()`:

The `avg()` method calculates the average value of a specific column. It is commonly used with numeric columns. Here's an example:



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

This query will calculate the average value of the `price` column in the `products` table, giving you the average price.

4. `max()`:

The `max()` method retrieves the maximum value from a specific column. It is frequently used with numeric or date/time columns. Here's an example:

```
$latestYear = DB::table('movies')->max('release_year');
```

This query will return the highest value from the `release_year` column in the `movies` table, representing the most recent year.

5. `min()`:

The `min()` method retrieves the minimum value from a specific column. It is commonly used with numeric or date/time columns. Here's an example:

```
$earliestDate = DB::table('events')->min('date');
```

This query will return the smallest value from the `date` column in the `events` table, representing the earliest date.

Answer to question no 12:

In Laravel's query builder, the `whereNot()` method is used to add a "where not" condition to a query. It allows me to exclude records that do not match a specified condition. This method is particularly useful when i want to filter out specific records from my result set.

The `whereNot()` method accepts two arguments: the column i want to compare and the value to compare against. It generates a SQL `WHERE` clause that excludes the records where the column's value matches the specified value.


Here's an example to demonstrate the usage of `whereNot()`:

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

In this example, the query builder will retrieve all the records from the `users` table except those where the `status` column is equal to `'inactive'`. It will generate the following SQL query:

```
SELECT * FROM users WHERE status <> 'inactive';
```


The `whereNot()` method can be used with other query builder methods to further refine my query. For example, i can combine it with the `where()` method to apply additional conditions:



```
$users = DB::table('users')
    ->where('age', '>', 18)
    ->whereNot('status', 'inactive')
    ->get();
```

In this case, the query will retrieve users who are older than 18 and have a status other than `'inactive'`.

The `whereNot()` method allows me to negate a specific condition in my query, providing flexibility in filtering out unwanted records from my result set.

Answer to question no 13:

In Laravel's query builder, the `exists()` and `doesntExist()` methods are used to check the existence of records in a table. They provide a convenient way to determine whether any records meet the specified conditions.

1. `exists()`:

The `exists()` method is used to check if there are any records that match the specified query conditions. It returns `true` if at least one record exists; otherwise, it returns `false`. Here's an example:



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

In this example, the `exists()` method checks if there are any active users in the `users` table. It will return `true` if there is at least one active user.

2. `doesn'tExist()`:


The `doesn'tExist()` method is the negation of `exists()`. It checks if no records exist that match the specified query conditions. It returns `true` if no records are found; otherwise, it returns `false`. Here's an example:



```
$noInactiveUsers = DB::table('users')  
    ->where('status', 'inactive')  
    ->doesn'tExist();
```

In this example, the `doesn'tExist()` method checks if there are no inactive users in the `users` table. It will return `true` if there are no inactive users.

Both methods can be used with other query builder methods to add additional conditions to the check. For example:



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

In this case, the `exists()` method is used to check if there are any active users with the role of "admin". It will return `true` if there is at least one active user with the "admin" role.

The `exists()` and `doesntExist()` methods are useful for performing conditional checks based on the existence of records in our database tables, allowing us to make decisions or perform actions based on the presence or absence of specific data.

Answer to question no 14:

Here's an example code snippet that retrieves records from the "posts" table where the "min_to_read" column is between 1 and 5 using Laravel's query builder:



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

print_r($posts);
```

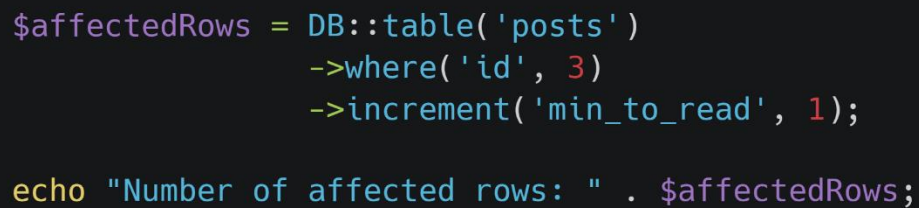
In this code, i use the `whereBetween()` method to specify the range for the "min_to_read" column. It takes two arguments: the column name and an array representing the range. In this case, i provide the range as `[1, 5]` to filter the records where "min_to_read" is between 1 and 5.

The `get()` method is used to execute the query and retrieve the result as a collection of objects representing the matched records.

Finally, the `print_r()` function is used to print the `$posts` variable, which will display the retrieved records from the "posts" table that satisfy the specified condition.

Answer to question no 15:

Here's an example code snippet that increments 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 and prints the number of affected rows:



```
$affectedRows = DB::table('posts')
    ->where('id', 3)
    ->increment('min_to_read', 1);

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

In this code, i use the `increment()` method to increase the value of the "min_to_read" column by 1 for the record with an "id" of 3. The first argument of the `increment()` method is the column to be incremented, and the second argument is the value by which it should be incremented.

The `where()` method is used to specify the condition for selecting the record with an "id" of 3.

The `increment()` method returns the number of affected rows, indicating how many records i updated. I store that value in the `$affectedRows` variable.

Finally, i print the number of affected rows using the `echo` statement, concatenating the `$affectedRows` variable with a message for clarity.