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

Ans: Laravel's Query Builder is a feature provided by the Laravel framework, which provides a convenient and expressive way to interact with databases. It allows developers to create database queries using a fluent and chainable API, making it easy to create complex SQL statements without writing raw SQL code.

Query Builder provides a set of methods that represent various SQL operations, such as selecting, inserting, updating, and deleting data. These methods can be chained together to create queries dynamically. For example, you can start with a basic SELECT statement and then add conditions, joins, and other clauses as needed.

One of the main advantages of using Query Builder is that it abstracts the underlying database system, allowing you to write database-agnostic code. Laravel supports multiple database systems including MySQL, PostgreSQL, SQLite and SQL Server. The query builder automatically generates the appropriate SQL syntax for the selected database, making your code portable and easily convertible between different database platforms.

The query builder provides a clean and readable syntax for defining conditions. You can use methods like where, orWhere, whereIn, and orWhereIn to specify different conditions in your query. This syntax allows you to express complex conditions using intuitive and logical code structures.

Additionally, the query builder offers features such as interesting loading relationships, pagination, and ordering results with minimal effort. You can load associated models with methods, order results using the PageNet method, and order results using the OrderB method, among others. These features help simplify common database operations and improve the overall development experience.

Another advantage of the query builder is that it provides protection against SQL injection attacks. It automatically parameterizes input values, ensures that user-supplied data is properly escaped, and prevents malicious database operations.

In summary, Laravel's Query Builder is a powerful tool that provides a simple and elegant way to interact with databases. It offers a fluent and chainable API, abstracts the database system, provides a clean syntax for conditions, and provides additional features for easy data manipulation. Using Query Builder, developers can write efficient, portable, and secure database queries with less effort and greater readability.

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.

Ans:

```
$posts = DB::table('posts')
```

```
    ->select('excerpt', 'description')
    ->get();


print_r($posts);
```

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

Ans: The **distinct()** method in Laravel's query builder is used to retrieve only unique values from a specified column or a set of columns in a database query. It ensures that the result set contains distinct (unique) values by removing any duplicate values.

The **distinct()** method is typically used in conjunction with the **select()** method. By chaining **distinct()** after **select()**, you can specify that you want to retrieve distinct values from the columns you select.

Here's an example:

$uniqueEmails = DB::table('users') ->select('email') ->distinct() ->get();

In the above code, we retrieve distinct email addresses from the "users" table. The **select('email')** method specifies that we only want to select the "email" column. The **distinct()** method ensures that only unique email addresses are retrieved. Finally, the **get()** method executes the query and retrieves the result.

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.

Ans: $posts = DB::table('posts')

```
        ->where('id', 2)
        ->first();


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

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.

Ans:

$posts = DB::table('posts')

```
        ->where('id', 2)
```

```
    ->pluck('description');


print_r($posts);
```

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

Ans: The **first()** and **find()** methods in Laravel's query builder are both used to retrieve single records from a table, but they have a slight difference in how they work.

The **first()** method retrieves the first record that matches the given query conditions. It returns a single object representing the first matching record. It is commonly used when you want to retrieve the earliest or oldest record based on a specific condition. Here's an example of using **first()** to retrieve the first record from the "users" table where the "id" is 1:

```
$user = DB::table('users') ->where('id', 1) ->first();
```

On the other hand, the **find()** method is used to retrieve a single record based on its primary key value. It directly fetches the record using the primary key value provided as an argument. It is commonly used when you know the specific primary key value you want to retrieve.

Here's an example of using **find()** to retrieve a record from the "users" table with the primary key value of 1:

```
$user = DB::table('users')  ->find(1);
```

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.

Ans:

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


print_r($posts);
```

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.

Ans:

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

```
    'min_to_read' => 2
]);

if ($result) {
    echo "Record inserted successfully.";
} else {
    echo "Failed to insert record.";
}
```

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.

Ans:

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

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

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.

Ans:

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

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

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.

Ans:

1. **count()**: This method is used to retrieve the number of rows returned by a query. It can be useful when you want to count the number of records that meet specific criteria. Here's an example:

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

   This will return the total count of rows in the "users" table.

2. **sum()**: The **sum()** method calculates the total sum of a specific column's values in a query result. It is commonly used to calculate the total value of a numeric column. Here's an example:

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

This will return the sum of the "amount" column in the "sales" table.

3. **avg()**: The **avg()** method calculates the average value of a specific column in a query result. It is often used to find the average value of a numeric column. Here's an example:

   $averageRating = DB::table('reviews')->avg('rating');

   This will return the average value of the "rating" column in the "reviews" table.

4. **max()**: The **max()** method retrieves the maximum value of a specific column in a query result. It is commonly used to find the highest value in a numeric column. Here's an example:

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

   This will return the maximum value of the "price" column in the "products" table.

5. **min()**: The **min()** method retrieves the minimum value of a specific column in a query result. It is used to find the lowest value in a numeric column. Here's an example:

   $lowestStock = DB::table('inventory')->min('stock');

   This will return the minimum value of the "stock" column in the "inventory" table.

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

Ans: 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 where a specific column does not match a given value or a set of values. The **whereNot()** method is typically used in combination with other query methods, such as **where()** or **orWhere()**, to create more complex conditions.

Here's an example of how to use the **whereNot()** method in Laravel's query builder:

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

In this example, the query will retrieve all records from the "users" table where the "status" column is not equal to "active". The **whereNot()** method takes two arguments: the column name and the value to compare against.

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?

1. Ans: **exists()**: The **exists()** method checks if there are any records that match the query criteria. It returns a boolean value, **true** if the records exist, and **false** otherwise. Here's an example:

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

   In this example, the **exists()** method is used to check if there are any records in the "users" table with a "status" column value of "active". If such records exist, it will return **true**; otherwise, it will return **false**.

2. **doesntExist()**: The **doesntExist()** method is the inverse of **exists()**. It checks if there are no records that match the query criteria. It returns a boolean

value, **true** if no records exist, and **false** if there are matching records. Here's an example:

```
$hasNoInactiveUsers =

 DB::table('users') ->where('status', 'inactive') ->doesntExist();
```

In this example, the **doesntExist()** method is used to check if there are no records in the "users" table with a "status" column value of "inactive". If no such records exist, it will return **true**; otherwise, it will return **false**.

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.

Ans:

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


print_r($posts);
```

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.

Ans:

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


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