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

Answer:

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in Laravel application and works perfectly with all of Laravel's supported database systems.

It allows developers to interact with databases using a fluent and expressive API, making it easier to retrieve, insert, update, and delete data.The query builder in Laravel abstracts the underlying database engine, allowing developers to write database queries in a database-agnostic manner. It supports multiple database systems, such as MySQL, PostgreSQL, SQLite, and SQL Server, among others.

The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings.

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

1. Fluent API: The query builder uses a fluent API, which means you can chain methods together to build complex queries in a readable and concise manner. This allows for a more expressive and intuitive way of constructing queries compared to writing raw SQL statements.

2. Parameter Binding: Laravel's query builder automatically handles parameter binding, which helps prevent SQL injection attacks. It securely binds user-supplied values to the query, ensuring that they are properly escaped and sanitized.

3. Query Building Methods: The query builder provides a rich set of methods for constructing various types of queries. These methods include selecting columns, joining tables, filtering records with conditions, grouping and ordering results, limiting and offsetting the number of records, and more. The query builder also supports advanced features like subqueries and unions.

4. Eloquent Integration: Laravel's query builder seamlessly integrates with the Eloquent ORM (Object-Relational Mapping) system, which is an ActiveRecord implementation in Laravel. This allows you to perform database operations using either the query builder or Eloquent, depending on your needs and preferences.

5. Database Abstraction: The query builder abstracts the underlying database engine, allowing you to write database-agnostic code. This means you can easily switch between different database systems without modifying your query code. Laravel's query builder takes care of translating the queries to the appropriate syntax for the chosen database.

Overall, Laravel's query builder provides a simple and elegant way to interact with databases by offering a fluent API, parameter binding for security, a wide range of query building methods, and seamless integration with the Eloquent ORM. It allows developers to write database queries more efficiently and maintainable, ultimately improving productivity and code quality.

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.

Answer:

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


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

foreach ($posts as $post) {
    echo "Excerpt: " . $post->excerpt . "<br>";
    echo "Description: " . $post->description . "<br>";
}

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?

Answer:

The distinct() method in Laravel's query builder is used to retrieve unique values from a column or a combination of columns in the result set. It ensures that duplicate values are eliminated, so each value appears only once in the result.

When used in conjunction with the select() method, the distinct() method allows you to specify which columns should be considered when determining uniqueness.

Here's an example to illustrate the usage of distinct() in conjunction with select():

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

In this example, we're querying the "users" table and selecting the "email" column. By applying the distinct() method, we ensure that only unique email addresses are returned in the result set.

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.

Answer:

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


if(is_null($post)) {
    return abort(404);
}

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.

Answer:

```
$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?

Answer:

first() retrieves the first record matching the query conditions. It returns an instance of the model or null if no matching record is found. It is commonly used when you want to retrieve a single record based on specific conditions.
Example usage of first():

find() retrieves a record by its primary key. It returns an instance of the model or null if no matching record is found. It is commonly used when you want to retrieve a record by its unique identifier.

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.

Answer:

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

Answer:

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

echo $result;
```

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.

Answer:

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

echo $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.

Answer:

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

echo $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.

Answer :

count(): Calculates the number of records matching the query.
Example:

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

sum(): Calculates the sum of a column's values.
Example:

```
$sum = DB::table('orders')->sum('total_amount');
echo $sum;
```

avg(): Calculates the average of a column's values.
Example:

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

max(): Retrieves the maximum value from a column.
Example:

```
$maxValue = DB::table('scores')->max('points');
echo $maxValue;
```

min(): Retrieves the minimum value from a column.
Example:

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

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

Answer:

The whereNot() method in Laravel's query builder is used to add a "not equal" condition to the query. It specifies that a column's value should not be equal to a given value. Here's an example:

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

In this example, the whereNot() method is used to retrieve all the users whose "status" column is not equal to 'inactive'. This method adds a "WHERE status <> 'inactive'" condition to the query.

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?

Answer:

The exists() and doesntExist() methods in Laravel's query builder are used to check the existence of records:

exists(): Returns true if records matching the query conditions exist, and false otherwise.
Example:

```
$exists = DB::table('users')->where('name', 'John')->exists();
echo $exists ? 'Records exist' : 'No records exist';
```

doesntExist(): Returns true if no records matching the query conditions exist, and false otherwise.
Example:

```
$doesntExist = DB::table('users')->where('name', 'John')->doesntExist();
echo $doesntExist ? 'No records exist' : 'Records exist';
```

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.

Answer :

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

Answer:

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