MODULE – 8(Migration)

1) How to do config database in laravel.

Laravel uses the PDO (PHP Data Objects) extension to connect to databases. In order to configure your database in Laravel, you will need to update the config/database.php file with your database credentials.

1. First, ensure you have a database set up and have the credentials (username, password, hostname, etc.) ready. Laravel supports several database systems, including MySQL, PostgreSQL, and SQLite.
2. Open the config/database.php file and find the connections array. This array contains all of the available connection configurations.
3. Choose a connection type from the array (e.g., mysql, sqlite, pgsql, etc.) and update the corresponding configuration options with your database credentials. For example, to configure a MySQL database, you might update the Default Database Connection Name:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

4. Open the .env file in the root directory of your Laravel project. Set the DB_CONNECTION variable in this file to match the connection type you chose in the previous step. For example, if you are using a MySQL database, you would set the DB_CONNECTION variable to mysql.

```
5. DB_CONNECTION=mysql
6. DB_HOST=127.0.0.1
7. DB_PORT=3306
8. DB_DATABASE=laravel
9. DB_USERNAME=root
   DB_PASSWORD=
```

10. Set the DB_HOST variable to the hostname of your database server. For example, if you are using a local development server, you might set this to localhost.
11. Set the DB_DATABASE variable to the name of your database.

12. Set the DB_USERNAME and DB_PASSWORD variables to the username and password for your database.
13. If you are using a MySQL or PostgreSQL database, you may also need to set the DB_PORT variable in your .env file to the correct port number for your database server.
14. If you are using a MySQL database, you may also need to set the DB_SOCKET variable in your .env file if your MySQL server uses a socket other than the default.
15. Save the .env file and exit.

Once you have configured your database connection, you can use the Laravel query builder or Eloquent ORM to perform CRUD operations on your database.

For example, to retrieve all rows from a table called users, you could use the following code:

Example:

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

2) Call MySQLi Store Procedure from Laravel.

1)Setup the Database Connection:

- Ensure you have the MySQLi extension enabled in your PHP configuration. You don't need to configure a separate database connection in Laravel for MySQLi because Laravel uses PDO by default. Instead, you can use the MySQLi extension directly.

2)Create a MySQL Stored Procedure:

- Create the MySQL stored procedure you want to call in your MySQL database. Here's an example:

```
DELIMITER //
CREATE PROCEDURE GetEmployees()
BEGIN
   SELECT * FROM employees;
END //
DELIMITER ;
```

Replace '**GetEmployees**' with your desired stored procedure name.

3)Call the Stored Procedure in Laravel:

- In your Laravel application, you can call the MySQL stored procedure using the MySQLi extension directly. Here's an example:

```
use mysqli;

$mysqli = new mysqli("localhost", "username", "password",
"database_name");

if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

// Call the stored procedure
$result = $mysqli->query("CALL GetEmployees()");

if ($result === false) {
    die("Stored procedure call failed: " . $mysqli->error);
}

// Process the results
while ($row = $result->fetch_assoc()) {
    // Handle each row of data
}

// Close the MySQLi connection
$mysqli->close();
```

Replace ‘**"localhost"**’, ‘**"username"**’, ‘**"password"**’, and ‘**"database_name"**’ with your database connection details.

4)Handle the Results:

- After calling the stored procedure, you can use a ‘**while**’ loop to iterate through the result set and process the data.

5)Close the MySQLi Connection:

- Always close the MySQLi connection when you are done to release resources.

Please note that while you can use MySQLi directly in Laravel, it's not the recommended approach for database interactions in Laravel applications. Laravel's built-in database features like Eloquent ORM and Query Builder provide a more convenient and secure way to interact with databases and are the

preferred methods in most cases. If possible, consider using Laravel's native database features instead.

3) Apply Curd Operation through Query Builder for Employee Management.

1)Create a Model:

Run the following Artisan command to create an Employee model and a migration:

```
php artisan make:model Employee –m
```

This will generate an '**Employee.php'** model file in the '**app'** directory and a migration file in the '**database/migrations'** directory.

2)Define the Schema:

In the generated migration file (e.g., '**create_employees_table.php'**), define the schema for the '**employees'** table. You can specify the columns you need. For example:

```
public function up()
{
    Schema::create('employees', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->string('phone');
        $table->timestamps();
    });
}
```

Run the migration to create the table:

```
php artisan migrate
```

3)Create Routes:

In the **routes/web.php** file, define routes for your CRUD operations. For example:

```
Route::resource('employees', 'EmployeeController');
```

This will create routes for all CRUD operations.

4)Create a Controller:
Generate a controller for your Employee model:

php artisan make:controller EmployeeController

5)Implement CRUD Operations in the Controller:

In the '**EmployeeController.php'** file, implement the CRUD operations using Laravel's Query Builder. Here's an example for each operation:

i)Create (Store):

```
public function store(Request $request)
{
   $data = $request->validate([
      'name' => 'required|string',
      'email' => 'required|email|unique:employees,email',
      'phone' => 'required|string',
   ]);

   DB::table('employees')->insert($data);
```

```
    return redirect()->route('employees.index')->with('success', 'Employee created successfully.');
}
```

ii)Read (Index and Show):

```
public function index()
{
    $employees = DB::table('employees')->get();

    return view('employees.index', compact('employees'));
}

public function show($id)
{
    $employee = DB::table('employees')->find($id);

    return view('employees.show', compact('employee'));
}
```

iii)Update (Edit and Update):

```
public function edit($id)
{
    $employee = DB::table('employees')->find($id);

    return view('employees.edit', compact('employee'));
}
```

```
public function update(Request $request, $id)
{
    $data = $request->validate([
        'name' => 'required|string',
        'email' => 'required|email|unique:employees,email,' . $id,
        'phone' => 'required|string',
    ]);

    DB::table('employees')->where('id', $id)->update($data);

    return redirect()->route('employees.index')->with('success', 'Employee updated successfully.');
}
```

iv)Delete (Destroy):

```
public function destroy($id)
{
    DB::table('employees')->where('id', $id)->delete();

    return redirect()->route('employees.index')->with('success', 'Employee deleted successfully.');
}
```

6)Create Views:

Create Blade views for displaying and managing employees. You can place these views in the '**resources/views/employees'** directory.

- **'index.blade.php'**: Display a list of employees.
- **'Show.blade.php'**: Display the details of a single employee.

- **‘create.blade.php’** and ‘**edit.blade.php’**: Create and edit employee forms.

7)Test Your CRUD Operations:

- Access your CRUD operations by visiting the appropriate routes, such as ‘**/employees’**, ‘**/employees/create’**, ‘**/employees/{id}/edit’**, etc., in your web browser.

That's it! You've implemented CRUD operations for Employee Management using Laravel's Query Builder. Be sure to customize the code and views to fit your specific requirements and styling.

4) Create All Migration for Employee management.

```
use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;


class CreateEmployeeManagerTable extends Migration

{

  public function up()

  {

    Schema::create('employee_manager', function (Blueprint $table) {

      $table->id();

      $table->unsignedBigInteger('employee_id');

      $table->unsignedBigInteger('manager_id');

      $table->timestamps();

    });

  }


  public function down()

  {
```

```
        Schema::dropIfExists('employee_manager');
    }
}
```