MODULE – 8(Migration)

1) How to do config database in laravel.

1) Installation of Laravel 9

   There are two TYPE

   Laravel Installer

Composer Create-ProjectJust Only Run one time in your pc

1)composer global require laravel/installer note: one time only

Then you can create project by belove command every time

2)laravel new project_name

Now open CMD & GO xampp

cd xampp/htdocs/ourfoldername

```
composer create-project laravel/laravel example-app
```

After Installation Run Project and default start in browser (localhost:8000)

Enter==> htdocs-> laravel_test- >php artisan serve

Config

The config folder includes various configurations and associated parameters required for the smooth functioning of a Laravel application. Various files included within the config folder are as shown in the image here. The filenames work as per the functionality associated with them.

We have lots of configuration in this folder like Database / Session / mail

Database

As the name suggests, this directory includes various parameters for database functionalities. It includes three sub-directories as given below −

- Seeds − This contains the classes used for unit testing databases. Means fake data
- Migrations − This folder helps in queries for migrating the database used in the web application.
- Factories − This folder is used to generate a large number of data records.

*Public

It is the root folder which helps in initializing the Laravel application. It includes the following files and folders −

- .htaccess − This file gives the server configuration.
- javascript and css − These files are considered as assets // all theme folder.
- **index.php** − This file is required for the initialization of a web application.This is first file in laravel load when project Run

*Resources : We load public file of theme and

Resources directory contains the files which enhances your web application. The sub-folders included in this directory and their purpose is explained below −

- assets − The assets folder includes files such as LESS and SCSS, that are required for styling the web application.
- lang − This folder includes configuration for localization or internalization.
- **views** − Views are the HTML files or templates which interact with end users and play a primary role in MVC architecture.We all our HTML page in this folder

Routes :

The `routes` directory contains all of the route definitions for your application. By default, several route files are included with Laravel: `web.php`, `api.php`, `console.php`, and `channels.php`.

Storage

This is the folder that stores all the logs and necessary files which are needed frequently when a Laravel project is running. The sub-folders included in this directory and their purpose is given below −

- app − This folder contains the files that are called in succession.
- framework − It contains sessions, cache and views which are called frequently.
- Logs − All exceptions and error logs are tracked in this sub folder.

Tests

All the unit test cases are included in this directory. The naming convention for naming test case classes is camel_case and follows the convention as per the functionality of the class.

Vendor

Laravel is completely based on Composer dependencies, for example to install Laravel setup or to include third party libraries, etc. The Vendor folder includes all the composer dependencies.

In addition to the above mentioned files, Laravel also includes some other files which play a primary role in various functionalities such as GitHub configuration, packages and third party libraries.

.env : All the laravel credential in this files and very importance // database connectivity

APP_KEY=base64:fmEr4jlgDbcM5mL9pNxZ6pjU9Y8IOTHYLuaUpYpr2Zo=

DB_CONNECTION=mysql

DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=hospital
DB_USERNAME=root
DB_PASSWORD=

Composer.json : All the configuration of laravel in this files and all dependency

Package.json : when we developing api and use in front and developer likes React and Angular and etc.. that times it used

Que : Where is laravel version packages and dependency and also how change ad after that what command fire in laravel

==========

Laravel connect database in two way

2) Config database .env and import DB class
3) Database connectivity by model

Config Database

Find **.env** file and config database connectivity

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel8
DB_USERNAME=root
DB_PASSWORD=
```

Note: If you don't find .env file then just search .env.example file and rename .env

Checkout Database

Above all configuration done from **config/database.php**

Import DB class in Controller

// ADD FOR USE ANY DATABASE QUERIES

use Illuminate\Support\Facades\DB;

Fetch Data From MySql

Function user()

```
{
        $data= DB::select("select * from users");
        return view('viewuserdb',['mydata'=>$data]);
}
```

user.blade.php

```
@foreach ($mydata as $items)
<tr>
        <td>{{$items->uid}}</td>
<td>{{$items->unm}}</td>
```

```
<td>{{$items->pass}}</td>
<td>{{$items->email}}</td>
<td>{{$items->mobile}}</td>

</tr>
@endforeach
```

Migration:

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The **Laravel `Schema` facade** provides database agnostic support for **creating** and **manipulating** tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

`php artisan config:Cache  : cache reconfig for our db connection`

==========

`php artisan make:migration create_posts_table`

php artisan make: migration create_posts_table --create=posts // with structure

Php artisan migrate

```
public function up()
    {
        Schema::create('clients', function (Blueprint $table) {
           $table->id();//$table->increments('id');
or
$table->id('custome_id');  // you can also add custom id

// it provide default created_at/ updated_at column by default
           $table->timestamps();



     $table->integer('votes');
$table->string('username'); // default size 255
$table->char('name', 100)->nullable();
```

```
      $table->char('name', 100)->index();
$table->char('name', 100)->unique();
$table->text('description1');
$table->longText('description');
$table->timeTz('sunrise', $precision = 0);
$table->dateTimeTz('created_at1', $precision = 0);
$table->double('amount1', 8, 2);
$table->float('amount', 8, 2);
$table->enum('status',['Block','Unblock'])->default('Unblock');
$table->bigInteger('mobile');

$table->unsignedBigInteger('cate_id');  // foregn key declare
      $table->foreign('cate_id')->references('id')->on('categories');


$table->foreign('user_id')->references('id')->on('users')-
>onDelete('cascade');


Or

Laravel 7
$table->foreignId('category_id')->constrained()->onDelete('cascade');

});

    }
```

Note : if any error occurs in migrate then follow below rules

Go => config/database.php remove mb4

==========

Add column after create table

php artisan make:migration create_countries_table
php artisan make:migration add_columns_to_countries_table

```
Then its create again new migration file for customer then add
remaining field in new generated migration file


$table->string('country',60)->nullable()->after('address');

$table->string('state',50)->nullable()->after('state');
```

php artisan migrate:make **add_columns_to_countries**

==========

`php artisan migrate:status` : like to see which migrations have run thus far

```
php artisan migrate --force
php artisan migrate:rollback   // only one roleback
php artisan migrate:rollback --step=4
```

`php artisan migrate:reset` : command will roll back all of your application's migrations

`php artisan migrate:refresh` : command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database: // rollback and fresh migration

`php artisan migrate:fresh` : The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

==========

For database connectivity you can use any DB software from

Config/database.php ===

Default mysql / **sqlite / pgsql / sqlsrv /**

DB Connectivity : Go in .env file set DB name

Then migration new connectivity new Catch

==========

php artisan make:migration users table name always with s
Php artisan make:model user model without s but same name

php artisan make:model user -m // make model with migration file

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.