STEP BY STEP API CRUD OPERATIOIN IN LARAVEL

Step 1: Setting up Laravel

```
composer create-project --prefer-dist laravel/laravel rest-api-crud
```

Step 2: Mysql database configuration

Create new database(newcurd) in mysql

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

Step 3: Create the Product Model with migration

```
>> php artisan make:model Product –a
```

Step 4: Migration

In database/migrations/YYYY_MM_DD_HHMMSS_create_products_table.php, update the up function to match the following.

```
public function up(): void
    {
        Schema::create('products', function (Blueprint $table) {
            $table->id();
            $table->string('name');

            $table->string('details');

            $table->timestamps();

        });

    }
```

Step 5: create product interface

Create a repository interface for the Product model. This separation allows for cleaner and more maintainable code.

```
php artisan make:interface /Interfaces/ProductRepositoryInterface
```

in the *Interfaces*, create a new file called Product*RepositoryInterface.php* and add the following code to it.

```
<?php

namespace App\Interfaces;

interface ProductRepositoryInterface
{
    public function index();
    public function getById($id);
    public function store(array $data);
    public function update(array $data,$id);
    public function delete($id);
}
```

Step 6: **create product respository class**

Create a repository class for the Product model.

```
php artisan make:class /Repositories/ProductRepository
```

in the classes, create a new file called Product*Repository.php* and add the following code to it.

```
<?php

namespace App\Repository;
use App\Models\Product;
use App\Interfaces\ProductRepositoryInterface;
class ProductReposiotry implements ProductRepositoryInterface
{
    public function index(){
        return Product::all();
    }

    public function getById($id){
       return Product::findOrFail($id);
    }

    public function store(array $data){
       return Product::create($data);
    }

    public function update(array $data,$id){
       return Product::whereId($id)->update($data);
    }

    public function delete($id){
       Product::destroy($id);
    }
}
```

Step 7: Bind the interface and the implementation

we need to do is bind ProductRepository to ProductRepositoryInterface.we do this via a Service Provider. Create one using the following command.

```
php artisan make:provider RepositoryServiceProvider
```

Open *app/Providers/RepositoryServiceProvider.php* and update the register function to match the following

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Interfaces\ProductRepositoryInterface;
use App\Repository\ProductReposiotry;
class RepositoryServiceProvider extends ServiceProvider
{
    /**
     * Register services.
     */
    public function register(): void
    {
        $this->app-
>bind(ProductRepositoryInterface::class,ProductReposiotry::class);
    }

    /**
     * Bootstrap services.
     */
    public function boot(): void
    {
        //
    }
}
```

Step 8: Request validation

we have two request StoreProductRequest and UpdateProductRequest add the following code to it.

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Http\Exceptions\HttpResponseException;
use Illuminate\Contracts\Validation\Validator;
class StoreProductRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string,
\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
     */
    public function rules(): array
    {
        return [
            'name' => 'required',
            'details' => 'required'
        ];
    }

    public function failedValidation(Validator $validator)
    {
        throw new HttpResponseException(response()->json([
            'success'   => false,
            'message'   => 'Validation errors',
            'data'      => $validator->errors()
        ]));
    }
}
```

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

use Illuminate\Http\Exceptions\HttpResponseException;

use Illuminate\Contracts\Validation\Validator;

class UpdateProductRequest extends FormRequest

{

    /**

     * Determine if the user is authorized to make this request.

     */

    public function authorize(): bool

    {

      return true;

    }

    /**

     * Get the validation rules that apply to the request.

     *

     * @return array<string, 

\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>

     */

    public function rules(): array

    {

        return [

            'name' => 'required',

            'details' => 'required'

        ];

    }

    public function failedValidation(Validator $validator)

    {

        throw new HttpResponseException(response()->json([

            'success'   => false,

            'message'   => 'Validation errors',
            'data'      => $validator->errors()

        ]));

    }

}
```

Step 9: **Common ApiResponseClass create**

This common response class is the best practice thing. Because you can response send con function use. Create one using the following command

```
php artisan make:class /Classes/ApiResponseClass
```

Add the following code to it.

```
<?php

namespace App\Classes;
use Illuminate\Support\Facades\DB;
use Illuminate\Http\Exceptions\HttpResponseException;
use Illuminate\Support\Facades\Log;
class ApiResponseClass
{
    public static function rollback($e, $message ="Something went wrong! Process not
completed"){
        DB::rollBack();
        self::throw($e, $message);
    }

    public static function throw($e, $message ="Something went wrong! Process not
completed"){
        Log::info($e);
        throw new HttpResponseException(response()->json(["message"=> $message], 500));
    }

    public static function sendResponse($result , $message ,$code=200){
        $response=[
            'success' => true,
            'data'    => $result
        ];
        if(!empty($message)){
            $response['message'] =$message;
        }
        return response()->json($response, $code);
    }

}
```

Step 10: create product resource

Create one using the following command.

```
php artisan make:resource ProductResource
```

Add the following code to it.

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class ProductResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' =>$this->id,
            'name' => $this->name,
            'details' => $this->details
        ];
    }
}
```

Step 11: **productcontroller class**

With our repository in place, let's add some code to our controller.
Open *app/Http/Controllers/ProductController.php* and update the code to match the following.

```
<?php

namespace App\Http\Controllers;

use App\Models\Product;
use App\Http\Requests\StoreProductRequest;
```

```php
use App\Http\Requests\UpdateProductRequest;
use App\Interfaces\ProductRepositoryInterface;
use App\Classes\ApiResponseClass;
use App\Http\Resources\ProductResource;
use Illuminate\Support\Facades\DB;
class ProductController extends Controller
{

    private ProductRepositoryInterface $productRepositoryInterface;

    public function __construct(ProductRepositoryInterface $productRepositoryInterface)
    {
        $this->productRepositoryInterface = $productRepositoryInterface;
    }
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        $data = $this->productRepositoryInterface->index();

        return ApiResponseClass::sendResponse(ProductResource::collection($data),'',200);
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(StoreProductRequest $request)
    {
        $details =[
            'name' => $request->name,
            'details' => $request->details
        ];
        DB::beginTransaction();
        try{
             $product = $this->productRepositoryInterface->store($details);

             DB::commit();
             return ApiResponseClass::sendResponse(new ProductResource($product),'Product
Create Successful',201);

        }catch(\Exception $ex){
            return ApiResponseClass::rollback($ex);
        }
    }
    /**
     * Display the specified resource.
     */
    public function show($id)
    {
        $product = $this->productRepositoryInterface->getById($id);

        return ApiResponseClass::sendResponse(new ProductResource($product),'',200);
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(Product $product)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(UpdateProductRequest $request, $id)
    {
        $updateDetails =[
            'name' => $request->name,
            'details' => $request->details
        ];
        DB::beginTransaction();
        try{
             $product = $this->productRepositoryInterface->update($updateDetails,$id);

             DB::commit();
             return ApiResponseClass::sendResponse('Product Update Successful','',201);

        }catch(\Exception $ex){
            return ApiResponseClass::rollback($ex);
        }
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy($id)
    {
         $this->productRepositoryInterface->delete($id);

        return ApiResponseClass::sendResponse('Product Delete Successful','',204);
    }
}
```

Step 12: Api route

Executing the subsequent command allows you to publish the API route file:

```
php artisan install:api
```

To map each method defined in the controller to specific routes, add the following code to *routes/api.php*.

```
<?php

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\ProductController;
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');


Route::apiResource('/products',ProductController::class);
```

>>   php artisan route:list

Now you can run and test this code by using postman