Course	:	Advanced Web Programming (PWL)
Study Program	:	D4 – Informatics Engineering / D4 – Business Information Systems
Semester	:	4 (four) / 6 (six)
Meeting to-	:	10 (ten)

JOBSHEET 10
RESTFUL API

Before we enter the material, we first create a new project that we will use to build a simple application with the topic *of Point of Sales (PoS),* according to the Case Study PWL.pdf. So we created a Laravel 10 project with the name

Before we enter the material, we first create a new project that we will use to build a simple application on the topic of Point of Sales (PoS), according to the PWL.pdf Case Study.
So we create a Laravel 10 project with the name PWL_POS.

We will use the PWL_POS project until the 12th meeting, as a project that we will study

A. RESTFUL API

Representational State Transfer (REST) is a style of software architecture that defines a set of principles for designing distributed application networks. A RESTful API is an application programming interface that follows the principles of REST to transfer data between client and server.

RESTful API is one of the architectures in the API (*Application Program Interface*) that uses HTTP requests to access data. Data is accessed using the GET, PUT, POST and DELETE HTTP methods which refer to read, update, create and delete operations on resources. In addition to HTTP methods, in RESTful or REST HTTP response is also used to define the response data returned. A commonly used response format is JSON (Javascript Object Notation).

B. JSON Web Token (JWT)

JWT stands for JSON Web Token. It is an open standard (RFC 7519) that defines a compact, self-contained token format for transferring claims between two parties. JWTs are often used in authentication and secure exchange of information in untrusted environments, such as the internet.

A JWT consists of three parts separated by periods ("."): header, payload, and signature. Each of these pieces consists of JSON data that is encrypted using a specific algorithm and then put together to form a complete token. The header contains the token type and the algorithm type used for encryption. The payload contains the claim or information you want to submit. The signature is used to verify that the token has not changed and that the data comes from a trusted source.

JWTs are often used in modern authentication and authorization systems, such as web applications and API web services, because of their flexibility in conveying encrypted information concisely.

We can use JWT to:

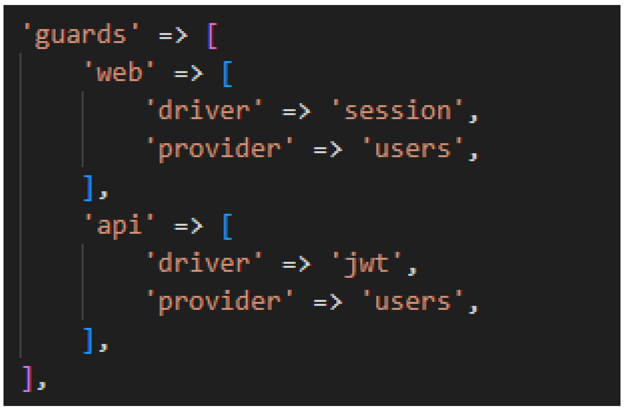
- Authentication
  When a user authenticates and gets a token, each subsequent request includes that token, allowing the user to access allowed routes, services, and resources.
- Exchange of information
  JSON Web Token is a good way to transmit information between parties securely. With the token that has been signed with the RSA algorithm, then we can know who made the request.

Here's how JWT works:

JWT (JSON Web Token) is a way to securely transfer information between two parties as JSON objects. It consists of three parts: header, payload, and signature. After the user successfully authenticates, the server generates a JWT token embedded in the HTTP request. The server then validates the token to grant access to the requested resource. It provides secure and stateless authentication without requiring storage of session state on the server.

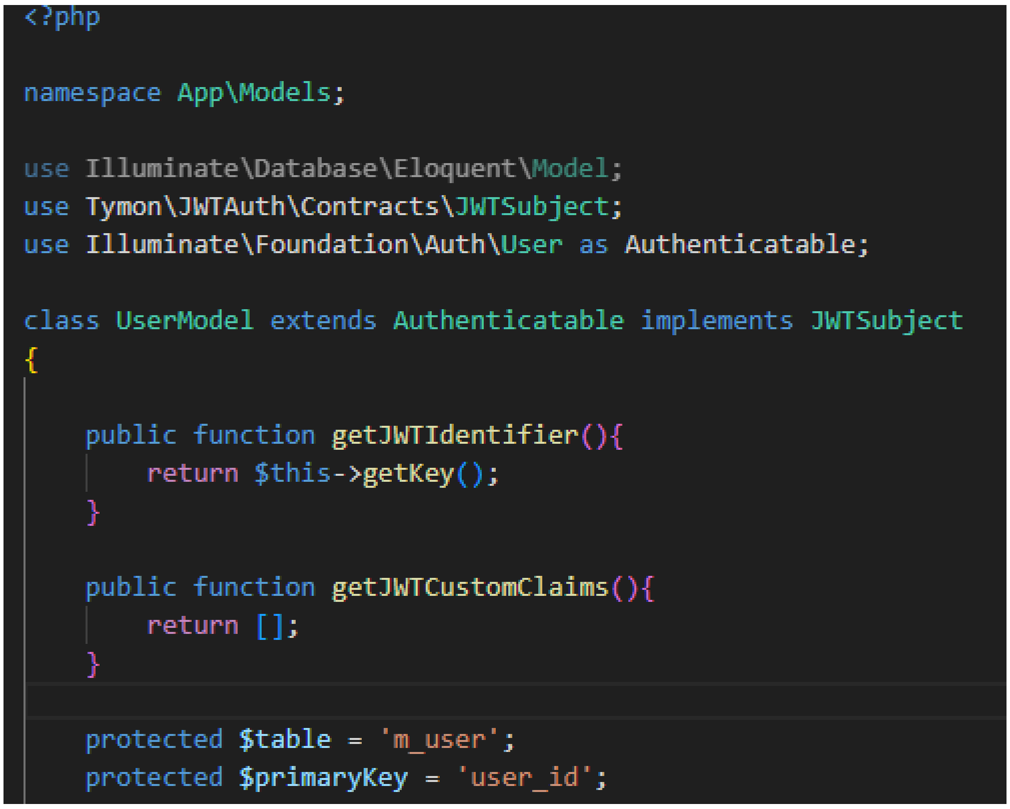
Practicum 1 – Creating a RESTful API Register

1. Before starting to create a REST API, first download the Postman application on https://www.postman.com/downloads.
   This application will be used to do all stages of practicum on this Jobsheet.
2. Perform the JWT installation by typing the following command:
   ```
   composer require tymon/jwt-auth:2.1.1
   ```
   Make sure you are connected to the internet.
3. After successfully installing JWT, continue to publish the configuration file with the following command:
   ```
   php artisan vendor:publish --
   provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
   ```
4. If the above command is successful, then we will get 1 new file which is config/jwt.php. In this file, configuration can be done if necessary.
5. After that, run the following request to create a JWT secret key.
   ```
   php artisan jwt:secret
   ```
   If successful, then the .env file will add a line containing the value of the JWT_SECRET key.
6. Next, configure the guard API. Open config/auth.php. Change the 'guards' section to something like this.



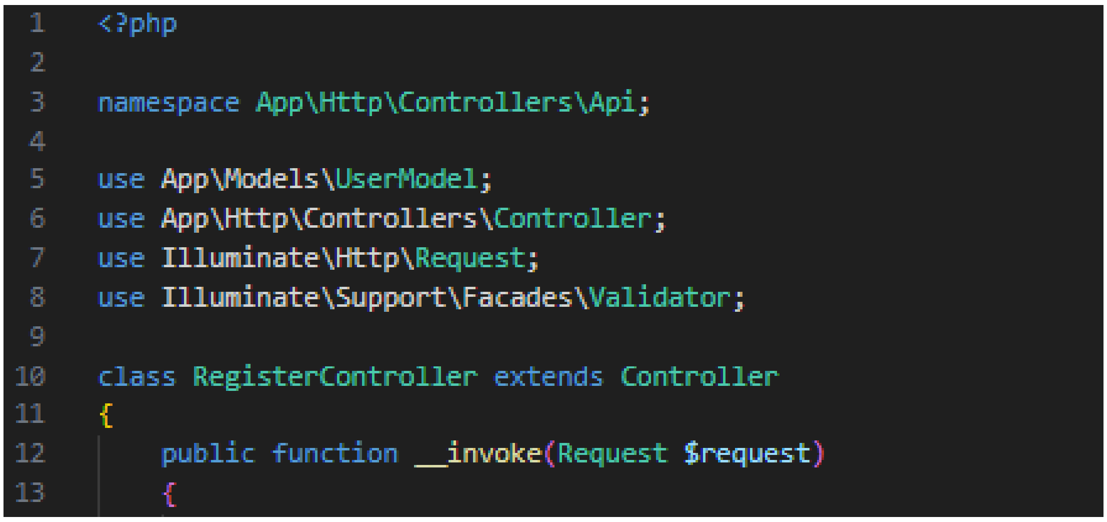
```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

7. We'll add code in the UserModel model, modify the code as follows:

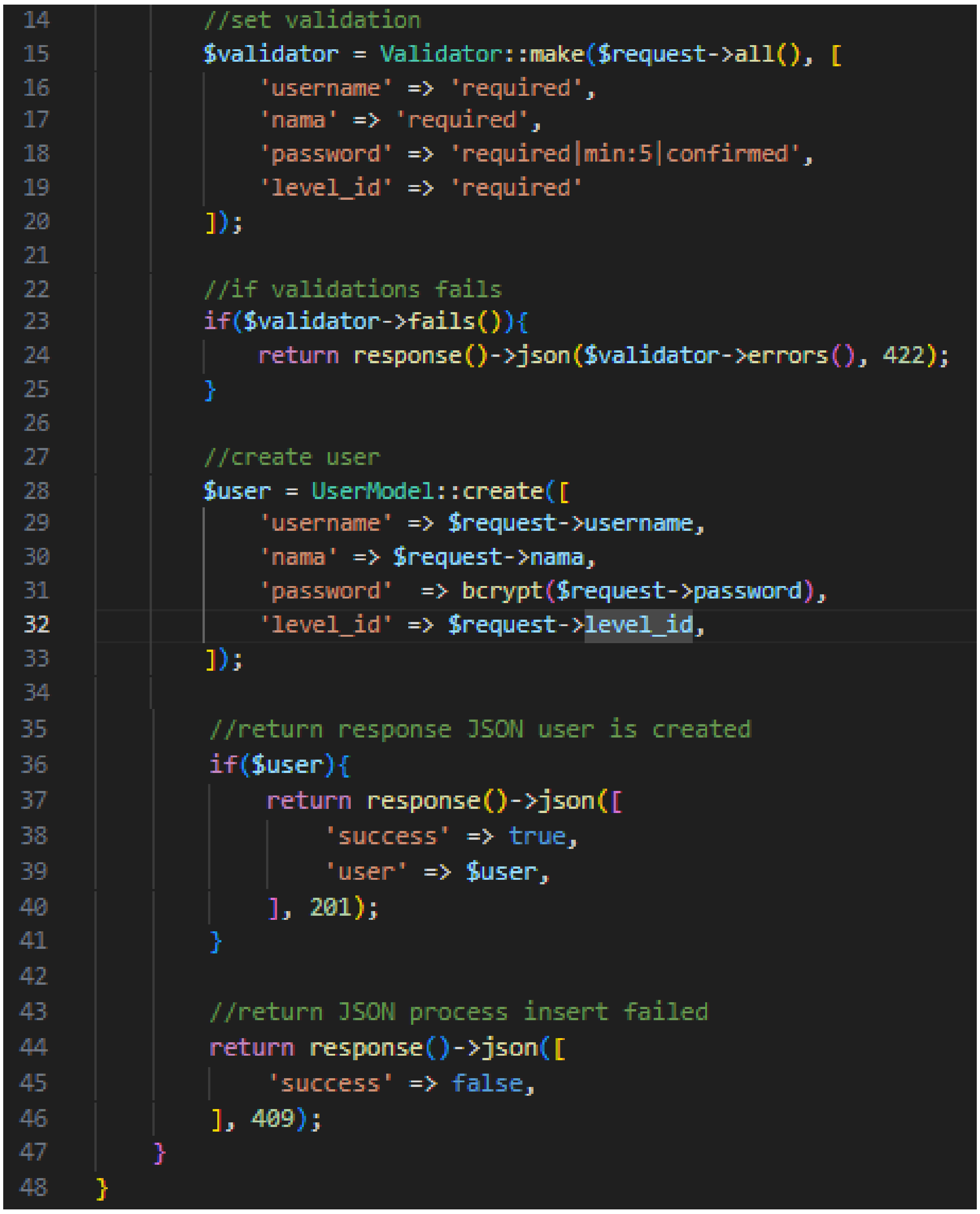


```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Tymon\JWTAuth\Contracts\JWTSubject;
use Illuminate\Foundation\Auth\User as Authenticatable;

class UserModel extends Authenticatable implements JWTSubject
{

    public function getJWTIdentifier(){
        return $this->getKey();
    }

    public function getJWTCustomClaims(){
        return [];
    }

    protected $table = 'm_user';
    protected $primaryKey = 'user_id';
```

8. Next we will create a controller to register by running the following request.
   `php artisan make:controller Api/RegisterController`
   If successful, there will be an additional controller in the Fire folder named RegisterController.
9. Open the file, and change the code to something like this.

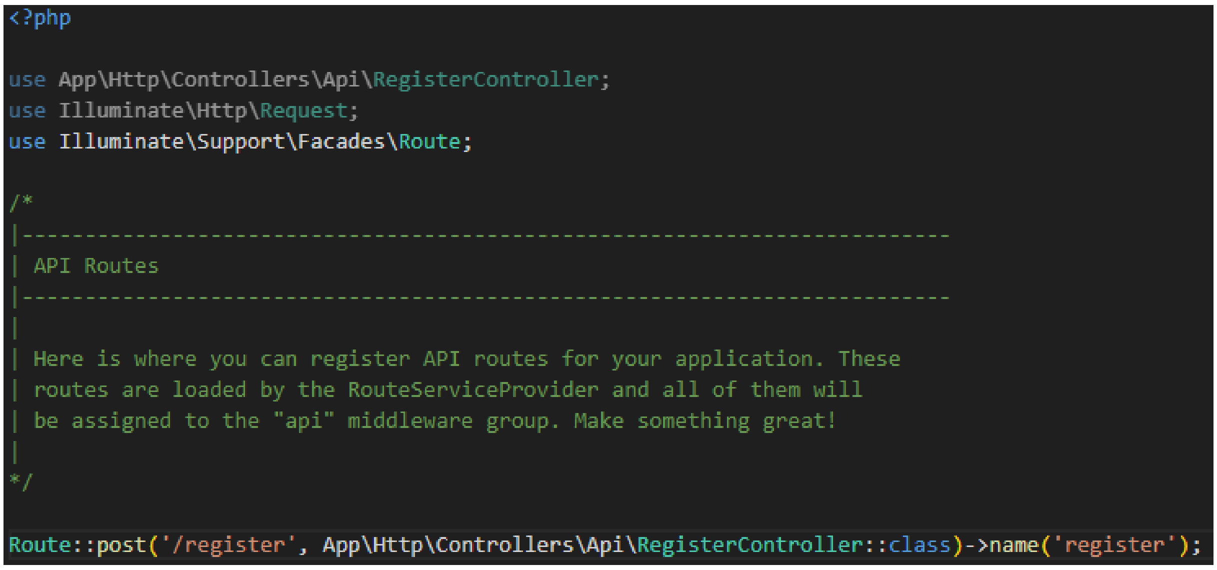


```php
<?php

namespace App\Http\Controllers\Api;

use App\Models\UserModel;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class RegisterController extends Controller
{
    public function __invoke(Request $request)
    {
```



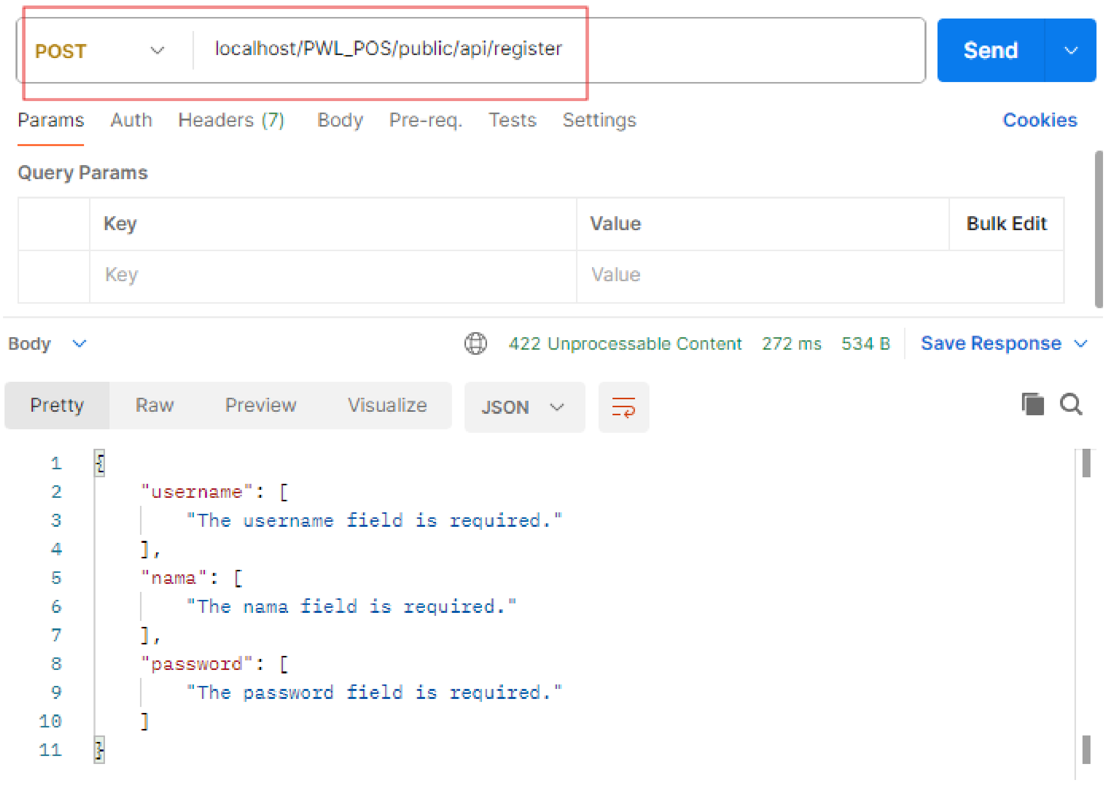
```php
        //set validation
        $validator = Validator::make($request->all(), [
            'username' => 'required',
            'nama' => 'required',
            'password' => 'required|min:5|confirmed',
            'level_id' => 'required'
        ]);

        //if validations fails
        if($validator->fails()){
            return response()->json($validator->errors(), 422);
        }

        //create user
        $user = UserModel::create([
            'username' => $request->username,
            'nama' => $request->nama,
            'password'  => bcrypt($request->password),
            'level_id' => $request->level_id,
        ]);

         //return response JSON user is created
         if($user){
             return response()->json([
                 'success' => true,
                 'user' => $user,
             ], 201);
         }

         //return JSON process insert failed
         return response()->json([
             'success' => false,
         ], 409);
    }
}
```

10. Next open routes/api.php, change all the code to as follows.



```php
<?php

use App\Http\Controllers\Api\RegisterController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "api" middleware group. Make something great!
|
*/

Route::post('/register', App\Http\Controllers\Api\RegisterController::class)->name('register');
```

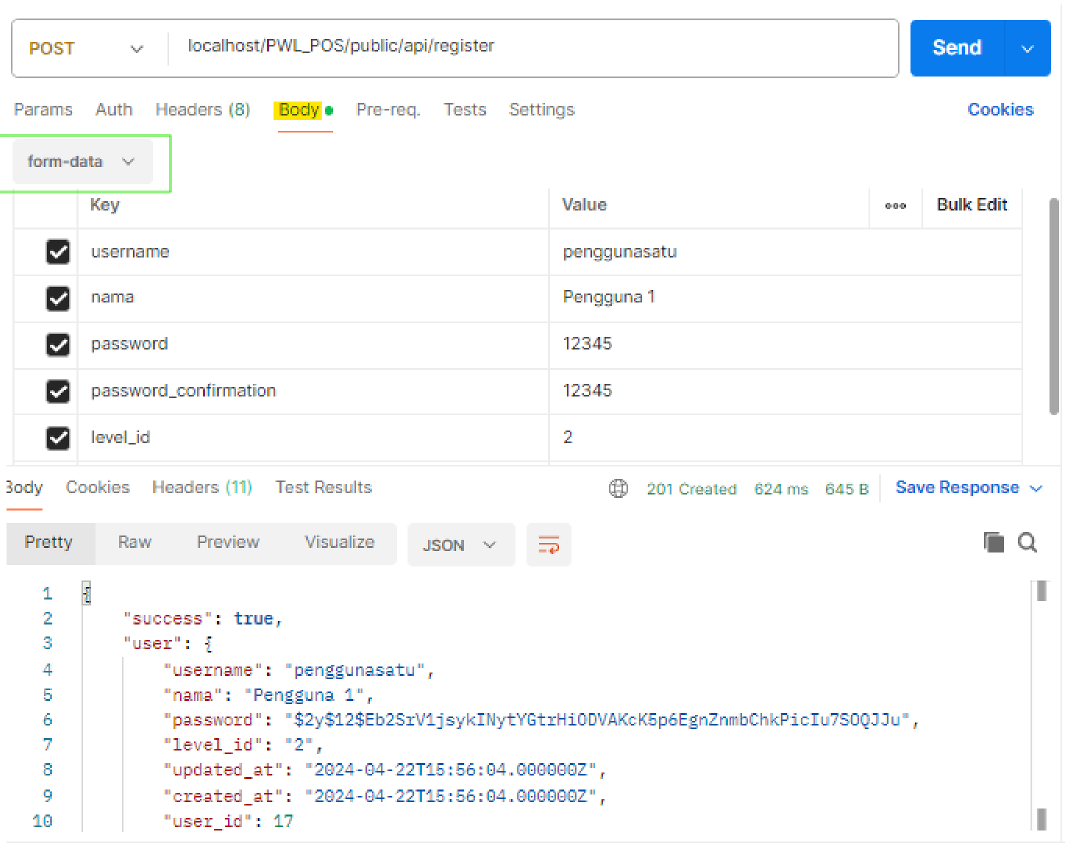
11. If so, we will test the REST API through the Postman application.
Open the Postman application, fill in the URL localhost/PWL_POS/public/api/register and the POST method. Click Send.



```
{
    "username": [
        "The username field is required."
    ],
    "nama": [
        "The nama field is required."
    ],
    "password": [
        "The password field is required."
    ]
}
```

If successful, a validation error will appear as shown above.
Do the same experiment and give a screenshot of the results of your experiment.

12. Now we try to enter the data. Click the Body tab and select form-data. Fill in the key according to the data column, and fill in the registration data using the value you want.



POST localhost/PWL_POS/public/api/register

Key	Value
username	penggunasatu
nama	Pengguna 1
password	12345
password_confirmation	12345
level_id	2

```
{
    "success": true,
    "user": {
        "username": "penggunasatu",
        "nama": "Pengguna 1",
        "password": "$2y$12$Eb2SrV1jsykINytYGtrHiODVAKcK5p6EgnZnmbChkPicIu7SOQJJu",
        "level_id": "2",
        "updated_at": "2024-04-22T15:56:04.000000Z",
        "created_at": "2024-04-22T15:56:04.000000Z",
        "user_id": 17
```

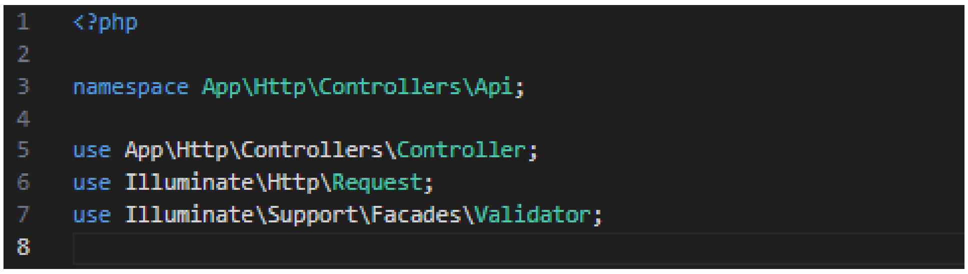
After clicking the Send button, if successful, a success message will come out as shown above.

Do the same experiment and give a screenshot of the results of your experiment.

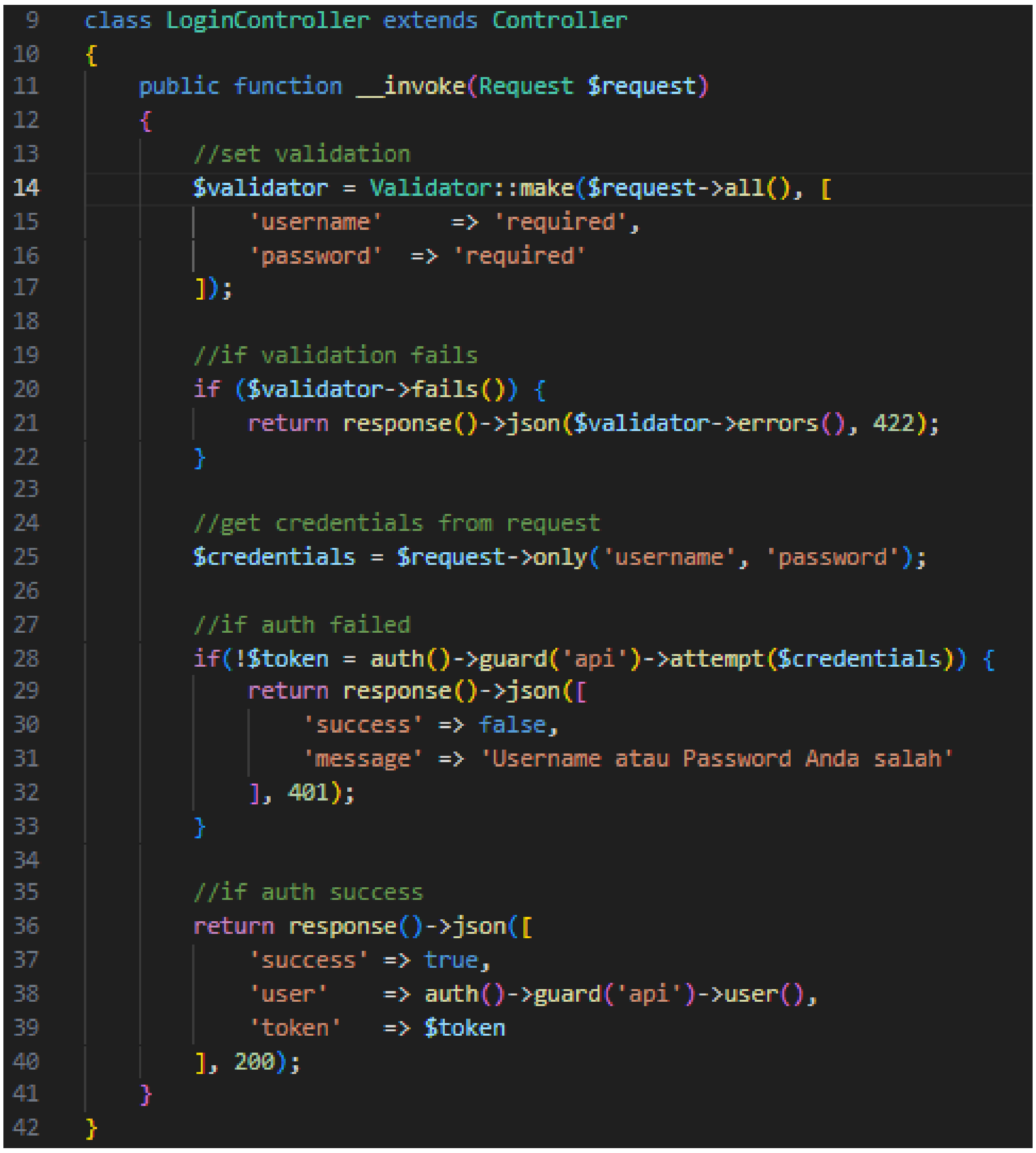
13. Commit file changes to Github.

Practicum 2 – Creating a RESTful API Login

1. We create a controller file named LoginController.
   `php artisan make:controller Api/LoginController`
   If successful, there will be an additional controller in the Fire folder named LoginController.
2. Open the file, and change the code to something like this.

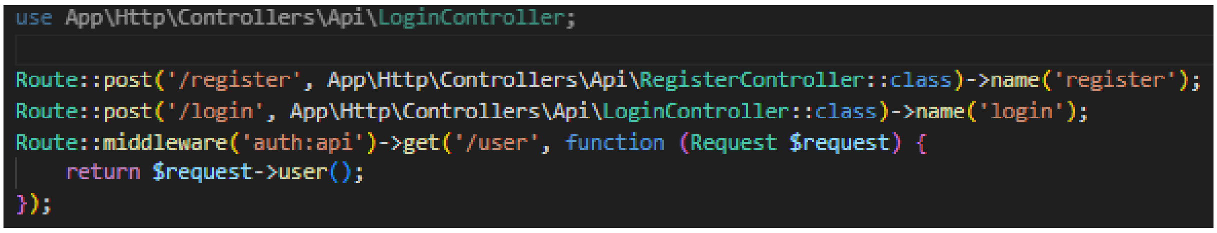


```php
<?php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

```



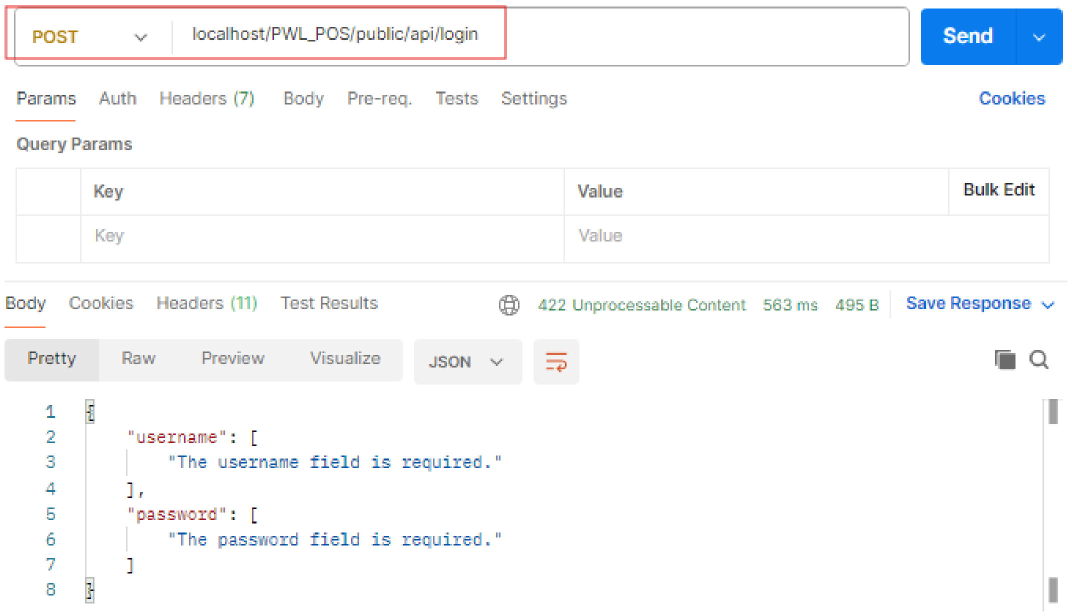
```php
class LoginController extends Controller
{
    public function __invoke(Request $request)
    {
        //set validation
        $validator = Validator::make($request->all(), [
            'username'     => 'required',
            'password'  => 'required'
        ]);

        //if validation fails
        if ($validator->fails()) {
            return response()->json($validator->errors(), 422);
        }

        //get credentials from request
        $credentials = $request->only('username', 'password');

        //if auth failed
        if(!$token = auth()->guard('api')->attempt($credentials)) {
            return response()->json([
                'success' => false,
                'message' => 'Username atau Password Anda salah'
            ], 401);
        }

        //if auth success
        return response()->json([
            'success' => true,
            'user'    => auth()->guard('api')->user(),
            'token'   => $token
        ], 200);
    }
}
```

3. Next, add new routes to the api.php file, namely /login and /user.



```php
use App\Http\Controllers\Api\LoginController;

Route::post('/register', App\Http\Controllers\Api\RegisterController::class)->name('register');
Route::post('/login', App\Http\Controllers\Api\LoginController::class)->name('login');
Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});
```

4. If so, we will test the REST API through the Postman application. Open the Postman application, fill in the URL localhost/PWL_POS/public/api/login and the POST method. Click Send.

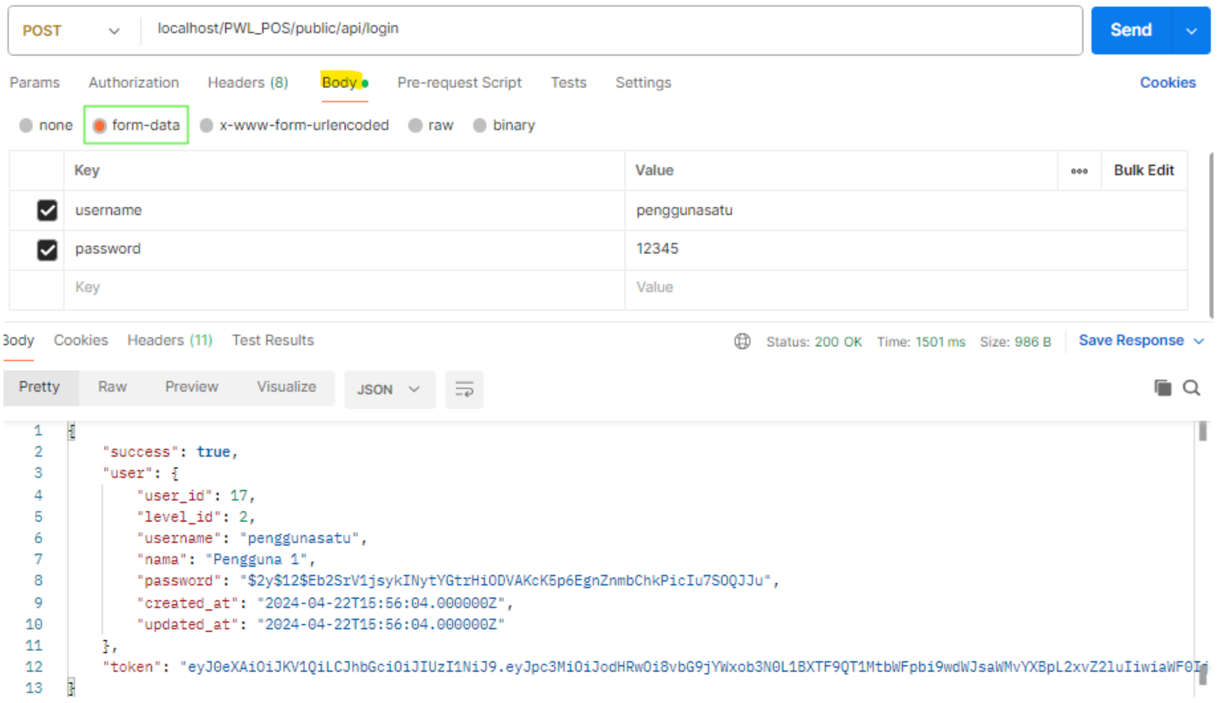


```
{
    "username": [
        "The username field is required."
    ],
    "password": [
        "The password field is required."
    ]
}
```

If successful, a validation error will appear as shown above.

Do the same experiment and give a screenshot of the results of your experiment.

5. Next, fill in the username and password according to the user data in the database. Click the Body tab and select form-data. Fill in the key according to the data column, and fill in the user data. Click the Send button, if successful it will display as follows. Copy the token value obtained at login because it will be required at logout.



	Key	Value
✓	username	penggunasatu
✓	password	12345
	Key	Value

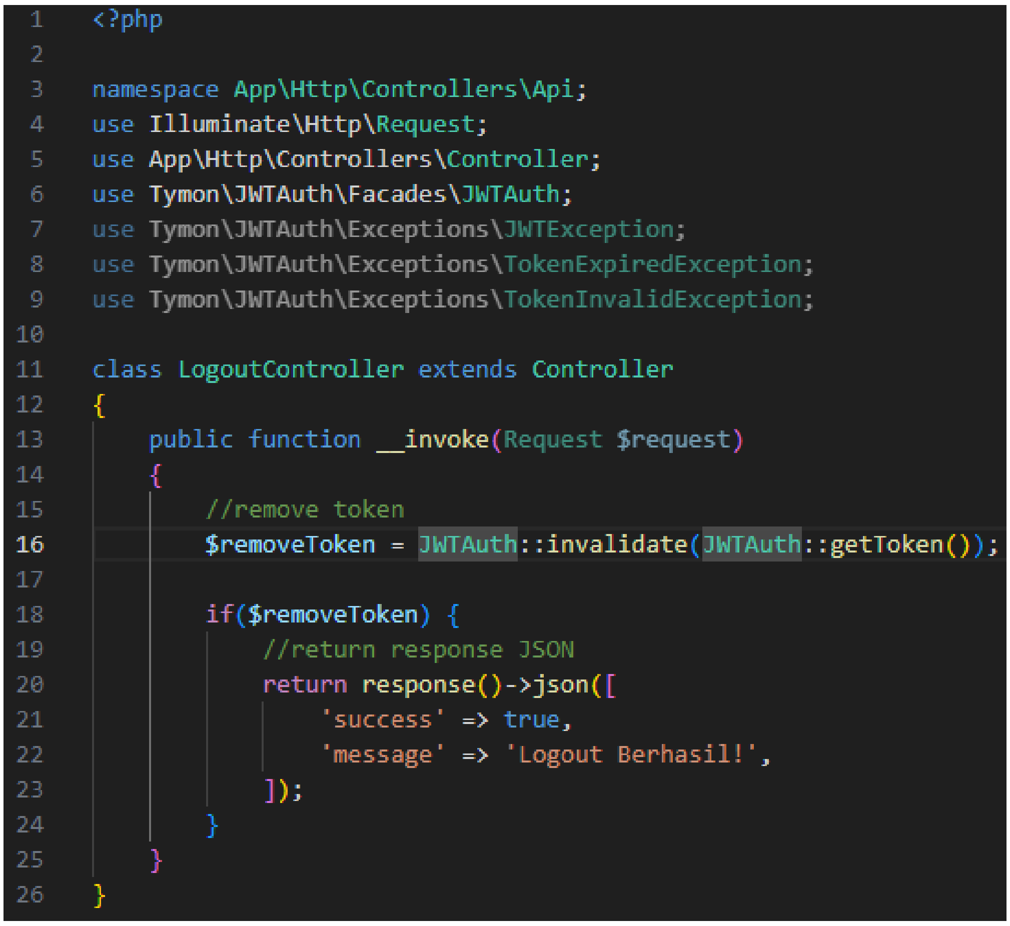
```
{
    "success": true,
    "user": {
        "user_id": 17,
        "level_id": 2,
        "username": "penggunasatu",
        "nama": "Pengguna 1",
        "password": "$2y$12$Eb2SrV1jsykINytYGtrHiODVAKcK5p6EgnZnmbChkPicIu7SOQJJu",
        "created_at": "2024-04-22T15:56:04.000000Z",
        "updated_at": "2024-04-22T15:56:04.000000Z"
    },
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0L1BXTF9QT1MtbWFpbi9wdWJsaWMvYXBpL2xvZ2luIiwiaWF0Ij
}
```

Do the same experiment and give a screenshot of the results of your experiment.

6. Do that experiment for incorrect data and provide a screenshot of your experiment results.
7. Try logging in with the correct data again. Now let's try to display the data of the logged in user using the URL localhost/PWL_POS/public/api/user and the GET method. Describe the results of the experiment.
8. Commit file changes to Github.

Practicum 3 – Create a RESTful API Logout

1. Add the following code to the .env file
   `JWT_SHOW_BLACKLIST_EXCEPTION=true`
2. Create a new Controller named LogoutController.
   `php artisan make:controller Api/LogoutController`
3. Open the file and change the code to something like this.

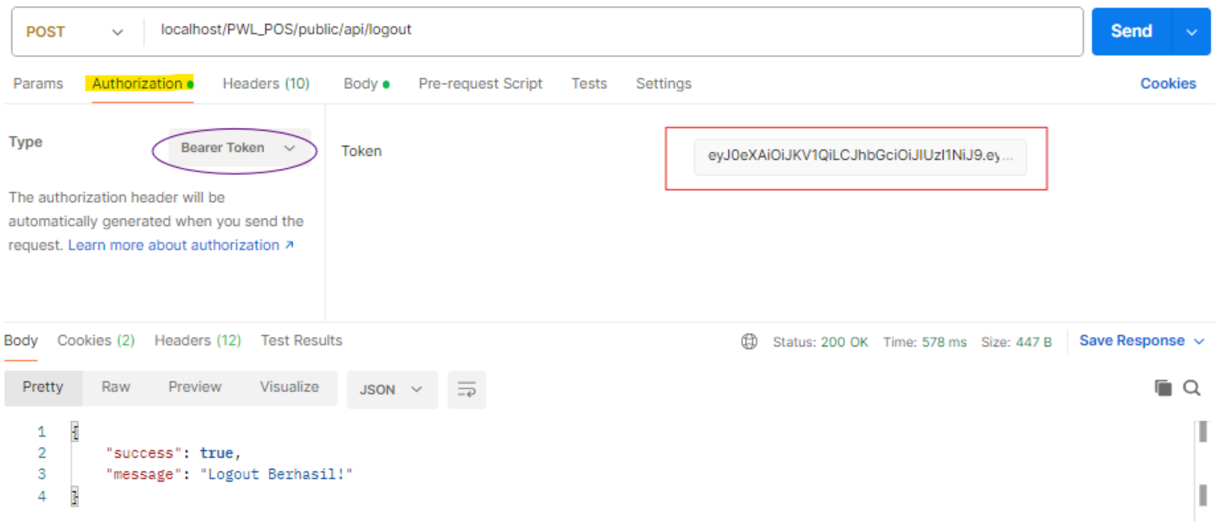


```php
<?php

namespace App\Http\Controllers\Api;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use Tymon\JWTAuth\Facades\JWTAuth;
use Tymon\JWTAuth\Exceptions\JWTException;
use Tymon\JWTAuth\Exceptions\TokenExpiredException;
use Tymon\JWTAuth\Exceptions\TokenInvalidException;

class LogoutController extends Controller
{
    public function __invoke(Request $request)
    {
        //remove token
        $removeToken = JWTAuth::invalidate(JWTAuth::getToken());

        if($removeToken) {
            //return response JSON
            return response()->json([
                'success' => true,
                'message' => 'Logout Berhasil!',
            ]);
        }
    }
}
```

4. Then we add routes to the api.php

```
Route::post('/logout', App\Http\Controllers\Api\LogoutController::class)->name('logout');
```

5. If so, we will test the REST API through the Postman application. Open the Postman application, fill in the URL localhost/PWL_POS/public/api/logout and the POST method.
6. Fill in the token on the Authorization tab, select Type which is Bearer Token. Fill in the token obtained when logging in. If so, click Send.


POST localhost/PWL_POS/public/api/logout
Type: Bearer Token
Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ey...
The authorization header will be automatically generated when you send the request. Learn more about authorization ↗
Status: 200 OK Time: 578 ms Size: 447 B

```
{
    "success": true,
    "message": "Logout Berhasil!"
}
```

Do the same experiment and give a screenshot of the results of your experiment.

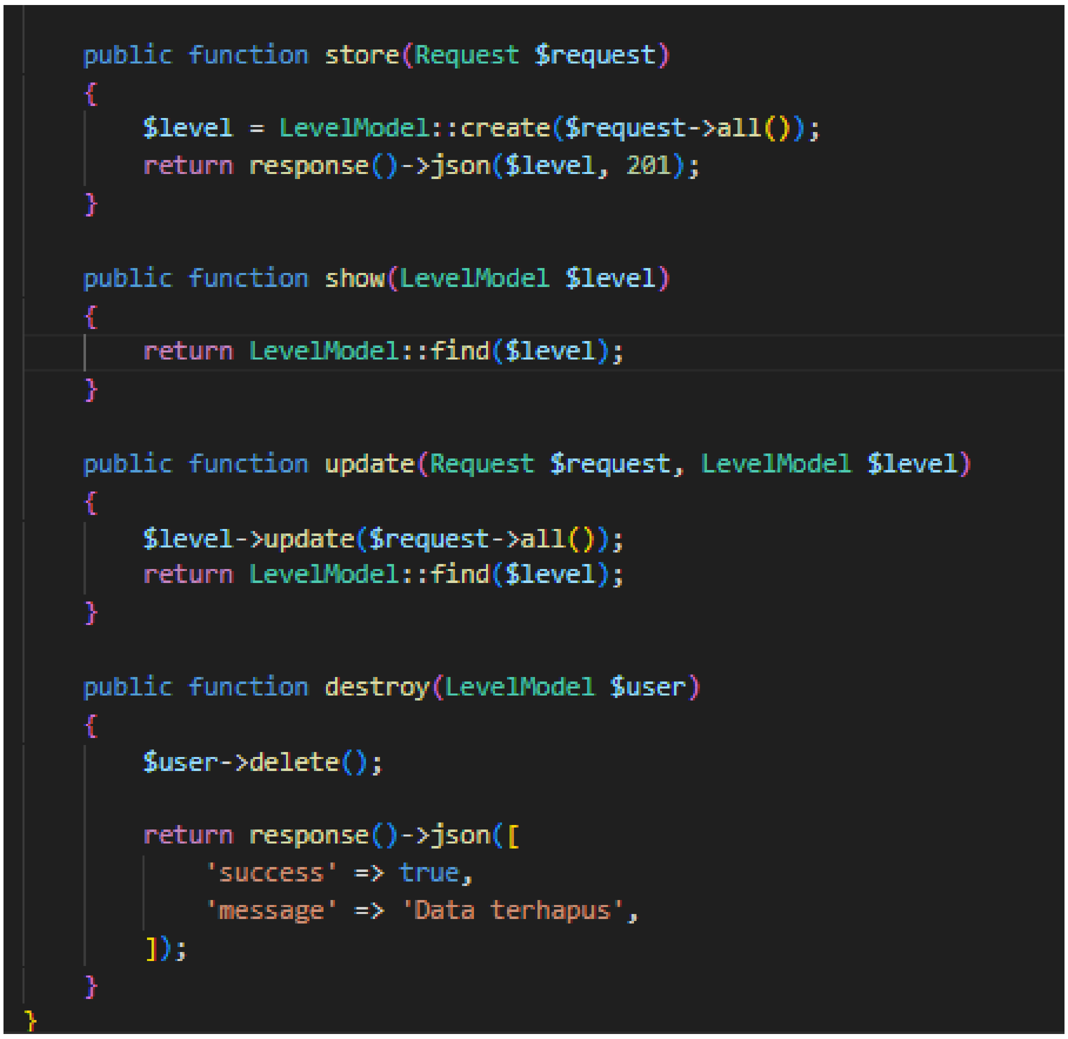
7. Commit file changes to Github.

Practicum 4 – CRUD implementation in RESTful API

In this practicum we will use m_level table to modify using the RESTful API.

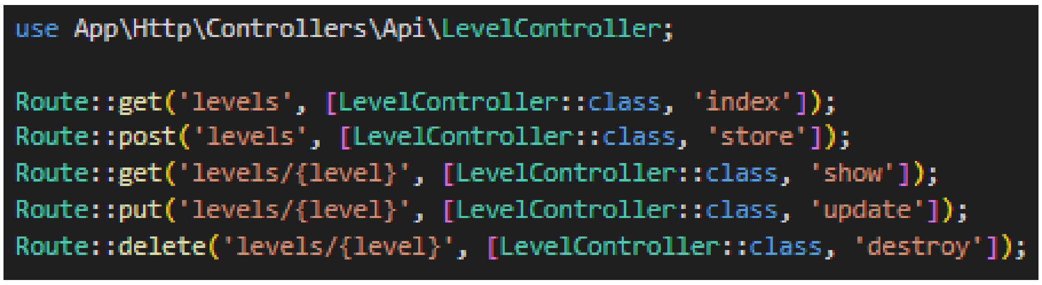
1. First, create a controller to process the API at the data level.
   `php artisan make:controller API/LevelController`
2. After success, open the file and write code like the following containing the CRUD function.

```
namespace App\Http\Controllers\Api;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\LevelModel;

class LevelController extends Controller
{
    public function index()
    {
        return LevelModel::all();
    }
```



```
    public function store(Request $request)
    {
        $level = LevelModel::create($request->all());
        return response()->json($level, 201);
    }

    public function show(LevelModel $level)
    {
        return LevelModel::find($level);
    }

    public function update(Request $request, LevelModel $level)
    {
        $level->update($request->all());
        return LevelModel::find($level);
    }

    public function destroy(LevelModel $user)
    {
        $user->delete();

        return response()->json([
            'success' => true,
            'message' => 'Data terhapus',
        ]);
    }
}
```

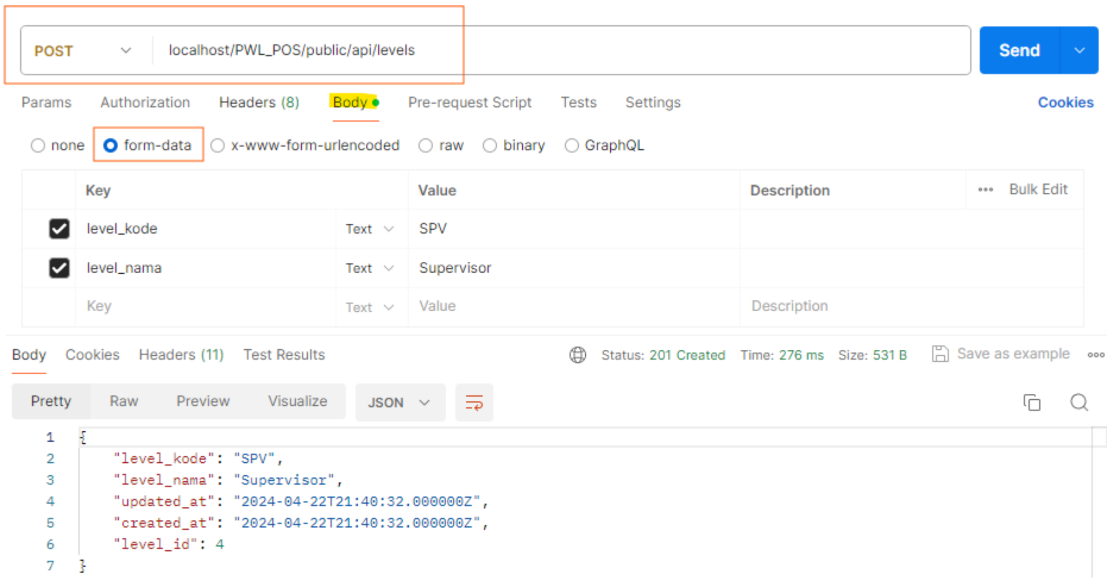
3. Then we complete the routes on api.php.



```
use App\Http\Controllers\Api\LevelController;

Route::get('levels', [LevelController::class, 'index']);
Route::post('levels', [LevelController::class, 'store']);
Route::get('levels/{level}', [LevelController::class, 'show']);
Route::put('levels/{level}', [LevelController::class, 'update']);
Route::delete('levels/{level}', [LevelController::class, 'destroy']);
```

4. If you have. Test the API starting from the function to display data. Use URLs: localhost/PWL_POS-main/public/api/levels and the GET method. Explain and provide screenshots of the results of your experiment.

5. Then, experiment adding data with URL: localhost/PWL_POS-main/public/api/levels and the POST method as below.



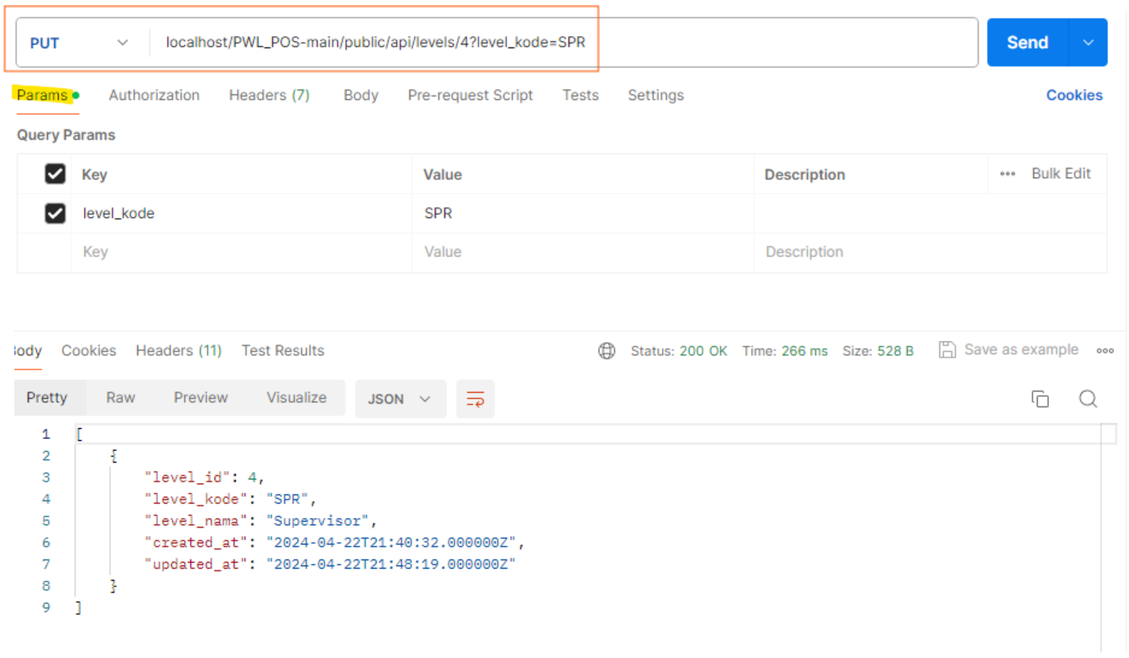
Key		Value	Description
level_kode	Text	SPV	
level_nama	Text	Supervisor	

```
{
    "level_kode": "SPV",
    "level_nama": "Supervisor",
    "updated_at": "2024-04-22T21:40:32.000000Z",
    "created_at": "2024-04-22T21:40:32.000000Z",
    "level_id": 4
}
```

Explain and provide screenshots of the results of your experiment.

6. Next, experiment displaying detailed data. Explain and provide screenshots of the results of your experiment.

7. If so, we try to edit the data using localhost/PWL_POS-main/public/api/levels/{id} and the PUT method. Fill in the data you want to change on the Param tab.



Key	Value	Description
level_kode	SPR	

```
[
    {
        "level_id": 4,
        "level_kode": "SPR",
        "level_nama": "Supervisor",
        "created_at": "2024-04-22T21:40:32.000000Z",
        "updated_at": "2024-04-22T21:48:19.000000Z"
    }
]
```

Explain and provide screenshots of the results of your experiment.

8. Finally, attempt to delete data. Explain and provide screenshots of the results of your experiment.

9. Commit file changes to Github.

ASSIGNMENT

Implement the CRUD API on other tables, namely tables m_user, m_kategori, and m_barang

That's it, and happy learning ***