Courses	:	Advanced Web Programming (PWL)
Study Program	:	D4 – Informatics Engineering / D4 – Business Information Systems
Semester	:	4 (four) / 6 (six)
Meeting to-	:	Three (3)

JOBSHEET 03
MIGRATION, SEEDER, DB FAÇADE, QUERY BUILDER, and ELOQUENT ORM

Previously we discussed about *Routing, Controller*, and *View* in Laravel. Before we get into making website-based applications, it would be nice for us to prepare a database as a place to store data in our application later. In addition, generally we need to prepare also the initial data that we use before creating an application, such as administrator user data, system settings data, etc.

For that, we need a technique to design/create a database table before creating the application. Laravel has features in database management such as, migration, seeders, models, etc.

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 **PWL_POS.**

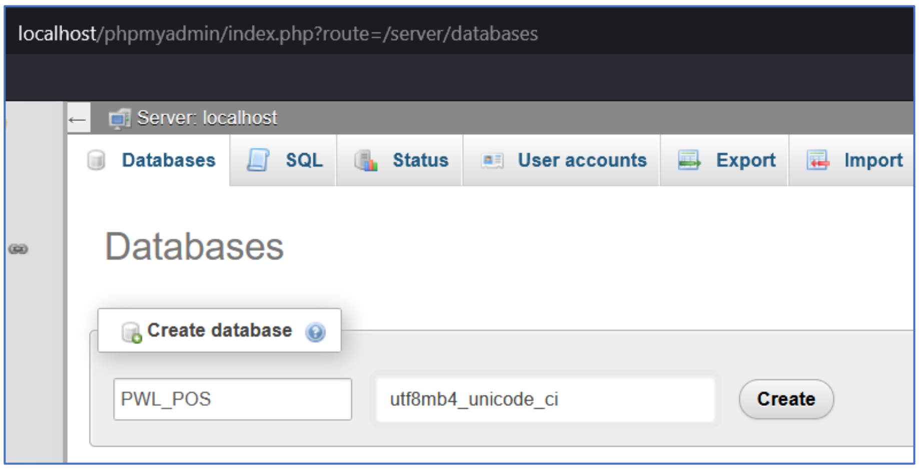
We will use PWL_POS project until the 12th meeting later, as a project that we will study

A. DATABASE SETTINGS

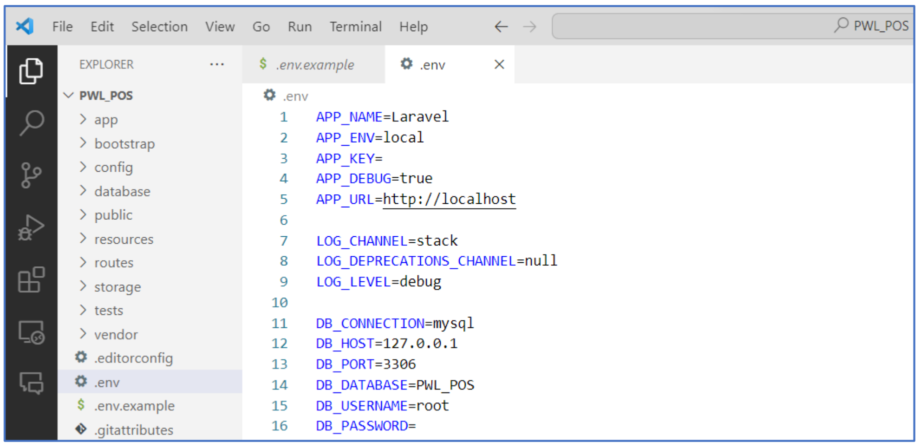
Database becomes an important component in building a system. This is because the database is a place to store transaction data on the system. We need to set the connection to the database to match the database we are using.

Practicum 1 - Database settings:

1. Open the phpMyAdmin application, and create a new database named `PWL_POS`

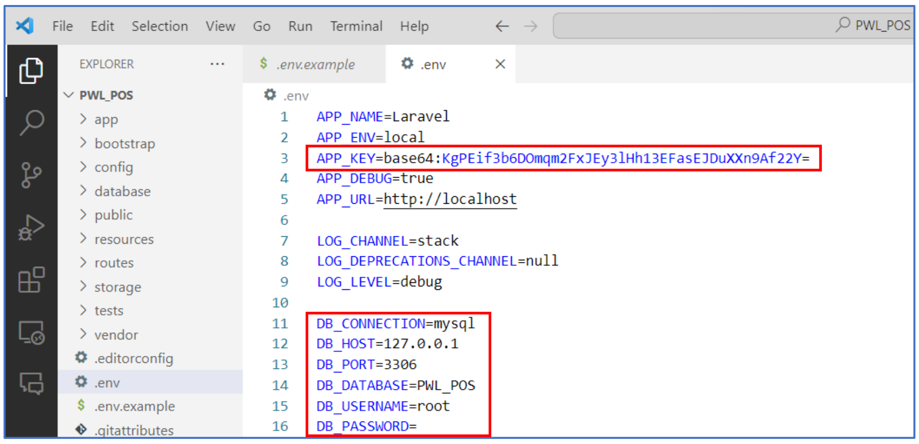


2. Open the VSCode application and open the PWL_POS project folder that we have created
3. Copy the `.env.example` file to `.env`
4. Open the `.env` file, and make sure **`the configuration APP_KEY`** valued. If it's not worth it, please *generate* it using `php artisan`.



```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

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

5. Edit the `.env file` and adjust it to the database that has been created



```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:KgPEif3b6DOmqm2FxJEy3lHh13EFasEJDuXXn9Af22Y=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

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

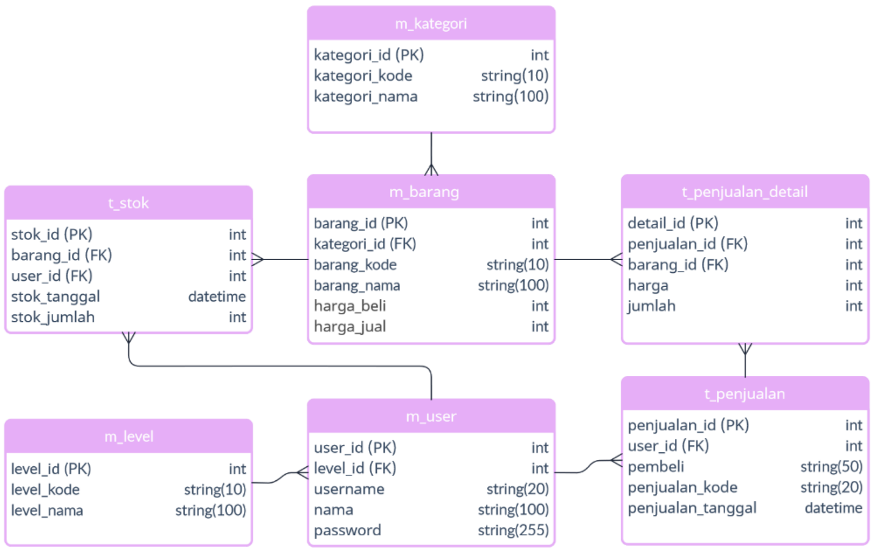
6. Report the results of this Practicum-1 and *commit* changes to *git*.

B. MIGRATION

Migration in Laravel is a feature that can help us manage databases efficiently using program code. Migration helps us create, edit, and delete table and column structures in the database that we have created quickly and easily. With Migration, we can also make changes to the database structure without having to delete existing data.

One of the advantages of using migration is to simplify the process of installing our application, when the application we create will be implemented on another server / computer.

In accordance with our learning topic to build a simple *Point of Sales (PoS)* system , then we need to create a migration according to the database design that has been defined in the `Case Study file PWL.pdf`



In creating a migration file in Laravel, what we need to pay attention to is the table structure we want to create.

TIPS *MIGRATION*

Create a migration file for tables that have no relationships (tables that do not have *foreign keys*) first, and continue by creating a migration file that has few relationships, and continue to a migration file with tables that have many relationships.

From the tips above, we can check for the existing database design by knowing the number of *foreign keys* that exist. And we can specify which table we will migrate first.

No	Table Name	Number of FK
1	`m_level`	0
2	`m_kategori`	0

3	`m_user`	1
4	`m_barang`	1
5	`t_penjualan`	1
6	`t_stok`	2
7	`t_penjualan_detail`	2

INFO

By default Laravel already has a `users` table to store user data, but in this practicum, we use the appropriate table from the `PWL.pdf Case Study file`, namely `m_user.`

Creating migration files can use 2 ways, namely

a. Use `artisan` to create *file migrations*

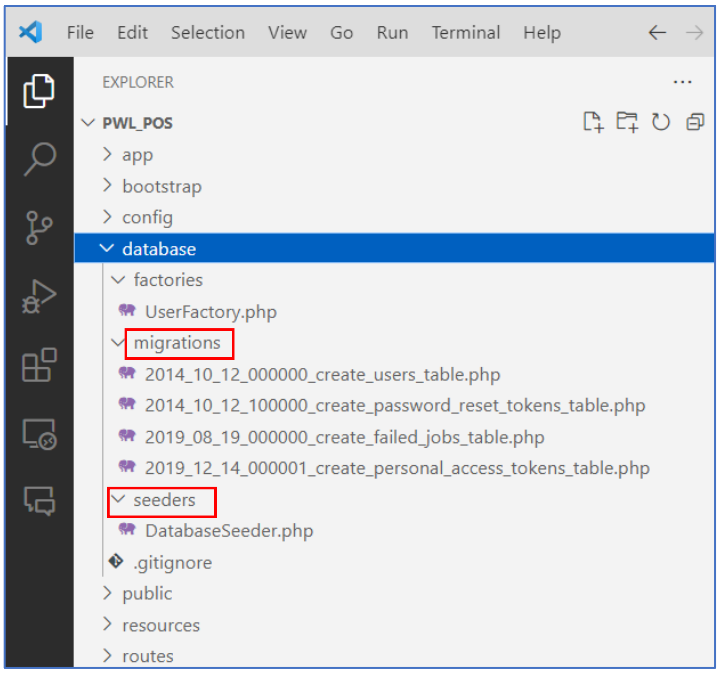
```
php artisan make:migration <nama-file-tabel> --create=<nama-tabel>
```

b. Use `artisan` to create *model files* + *migration files*

```
php artisan make:model <nama-model> -m
```

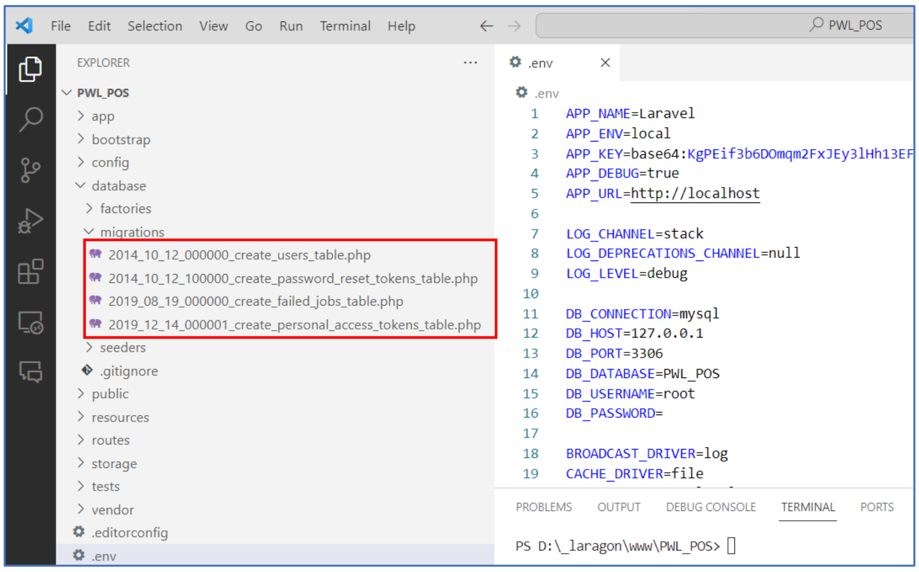
The `-m` command above is *a shorthand* for the option of creating a created model-driven migration file.

In Laravel, *migration* files or *seeders* are located in the `PWL_POS/database folder`


File
Edit
Selection
View
Go
Run
Terminal
Help
EXPLORER
PWL_POS
app
bootstrap
config
database
factories
UserFactory.php
migrations
2014_10_12_000000_create_users_table.php
2014_10_12_100000_create_password_reset_tokens_table.php
2019_08_19_000000_create_failed_jobs_table.php
2019_12_14_000001_create_personal_access_tokens_table.php
seeders
DatabaseSeeder.php
.gitignore
public
resources
routes

Practicum 2.1 - Unrelated migration file creation

1. Open your VSCode terminal, for the one in the red box is the default of laravel

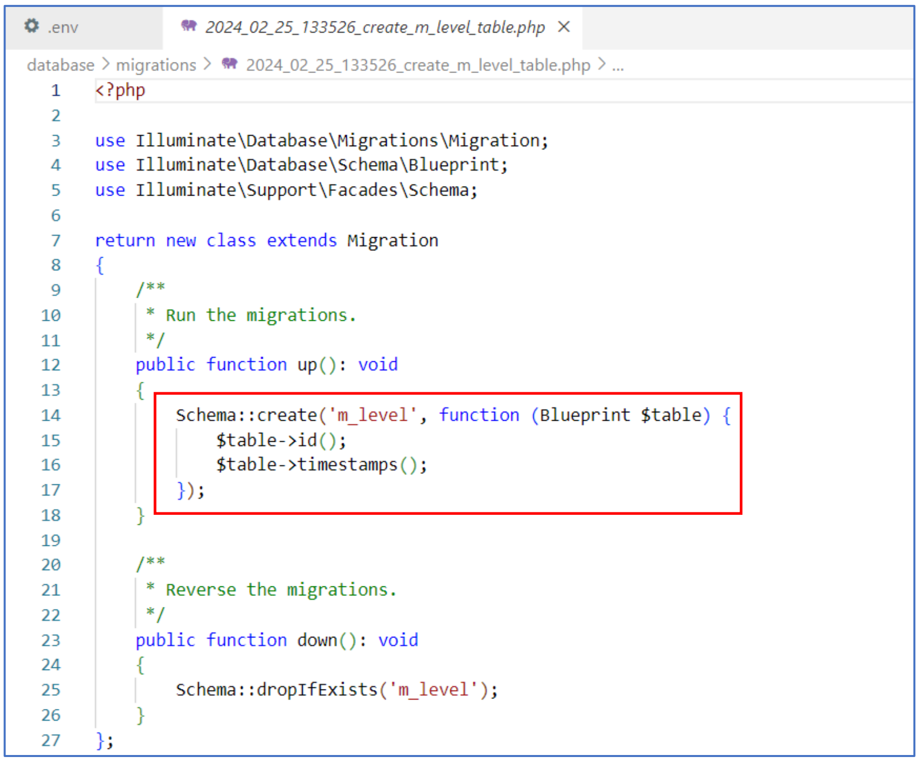

File Edit Selection View Go Run Terminal Help
PWL_POS
EXPLORER
PWL_POS
app
bootstrap
config
database
factories
migrations
2014_10_12_000000_create_users_table.php
2014_10_12_100000_create_password_reset_tokens_table.php
2019_08_19_000000_create_failed_jobs_table.php
2019_12_14_000001_create_personal_access_tokens_table.php
seeders
.gitignore
public
resources
routes
storage
tests
vendor
.editorconfig
.env
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:KgPEif3b6DOmqm2FxJEy3lHh13EF
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=PWL_POS
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
```

```
PS D:\_laragon\www\PWL_POS>
```

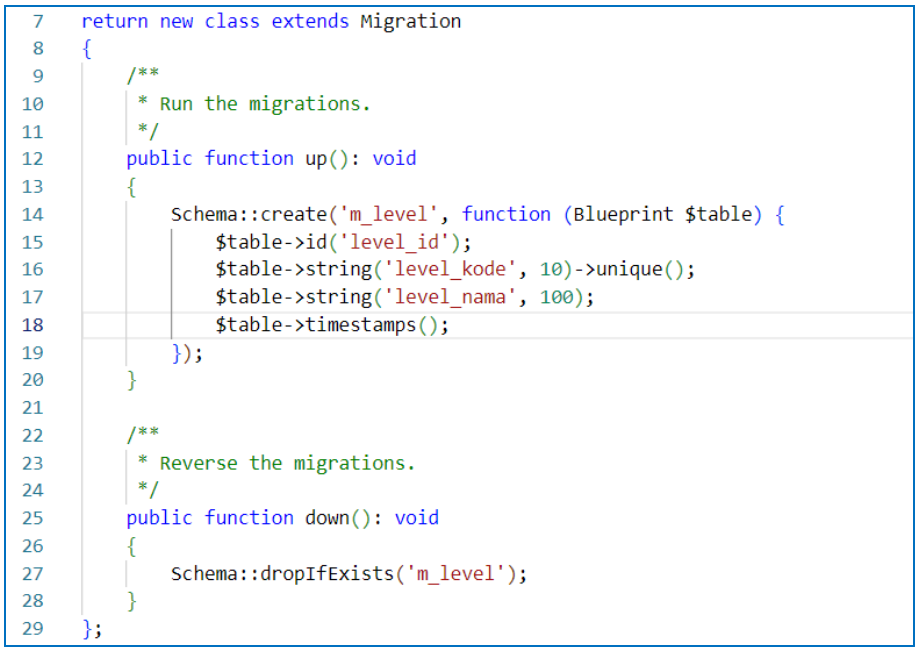
2. We ignore the one in the red box first (don't delete it)
3. We create a migration file for the m_level table with the command

```
php artisan make:migration create_m_level_table --create=m_level
```


.env
2024_02_25_133526_create_m_level_table.php
database > migrations > 2024_02_25_133526_create_m_level_table.php > ...

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('m_level', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('m_level');
    }
};
```

4. We pay attention to the part in the red box, the part that we will modify according to the existing database design



```php
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('m_level', function (Blueprint $table) {
            $table->id('level_id');
            $table->string('level_kode', 10)->unique();
            $table->string('level_nama', 100);
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('m_level');
    }
};
```

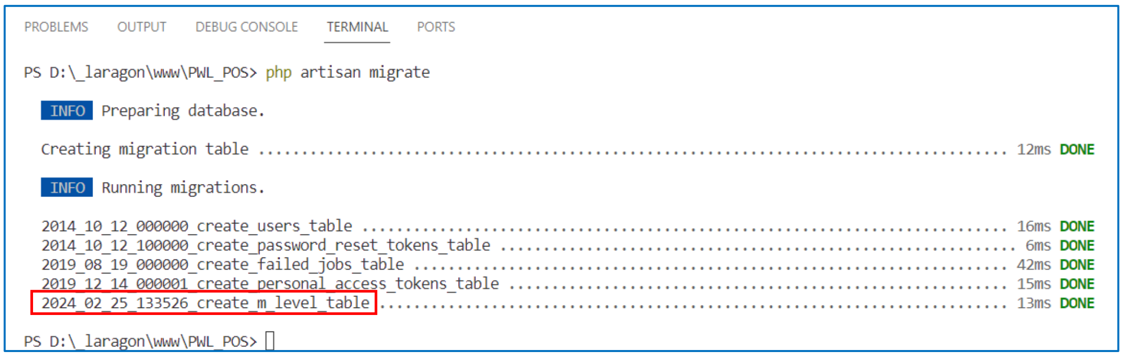
INFO

In Laravel's migration feature, there are various functions to create columns in the database table. Please check here

`https://laravel.com/docs/10.x/migrations#available-column-types`

5. Save the code in step 4, then run this command in the VSCode terminal to migrate

```
php artisan migrate
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\_laragon\www\PWL_POS> php artisan migrate

  INFO  Preparing database.

  Creating migration table ............................................................................... 12ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ................................................................... 16ms DONE
  2014_10_12_100000_create_password_reset_tokens_table .................................................... 6ms DONE
  2019_08_19_000000_create_failed_jobs_table ............................................................. 42ms DONE
  2019_12_14_000001_create_personal_access_tokens_table .................................................. 15ms DONE
  2024_02_25_133526_create_m_level_table ................................................................. 13ms DONE

PS D:\_laragon\www\PWL_POS> 
```

6. Then we check in phpMyAdmin whether the table has been generated or not

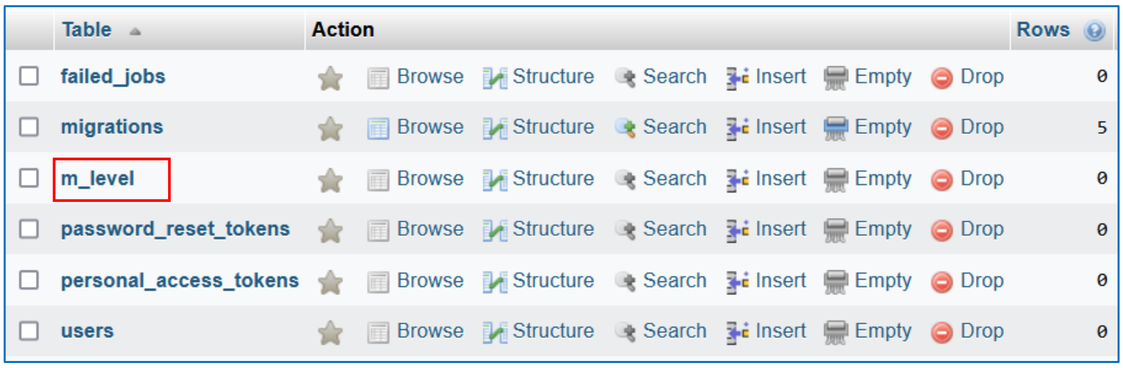


	Table	Action							Rows
☐	failed_jobs	★	Browse	Structure	Search	Insert	Empty	Drop	0
☐	migrations	★	Browse	Structure	Search	Insert	Empty	Drop	5
☐	m_level	★	Browse	Structure	Search	Insert	Empty	Drop	0
☐	password_reset_tokens	★	Browse	Structure	Search	Insert	Empty	Drop	0
☐	personal_access_tokens	★	Browse	Structure	Search	Insert	Empty	Drop	0
☐	users	★	Browse	Structure	Search	Insert	Empty	Drop	0

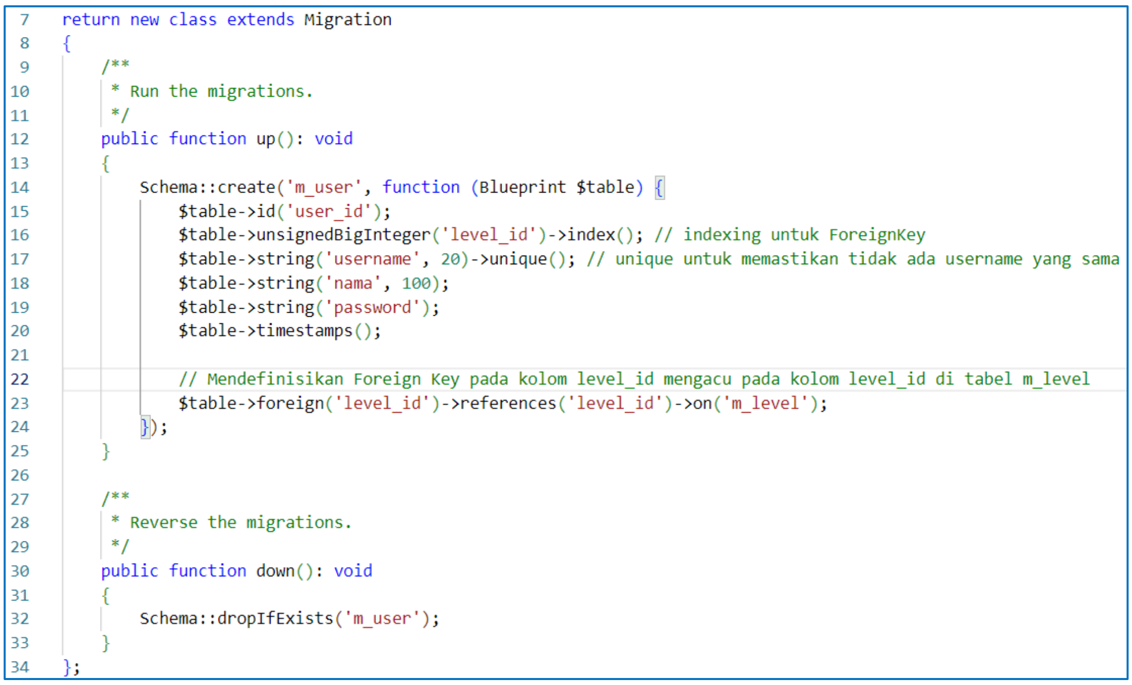
7. Ok, the table has been created in the database
8. Create a *database* table with *migration* for m_kategori tables that both have no *foreign keys*
9. Report the results of this Practicum-2.1 and *commit* changes to *git*.

Practicum 2.2 - Creation of migration files with relationships

1. Open your VSCode terminal, and create a migration file for the m_user `table`

```
php artisan make:migration create_m_user_table --table=m_user
```

2. Open the migration file for the m_user table, and modify it as follows



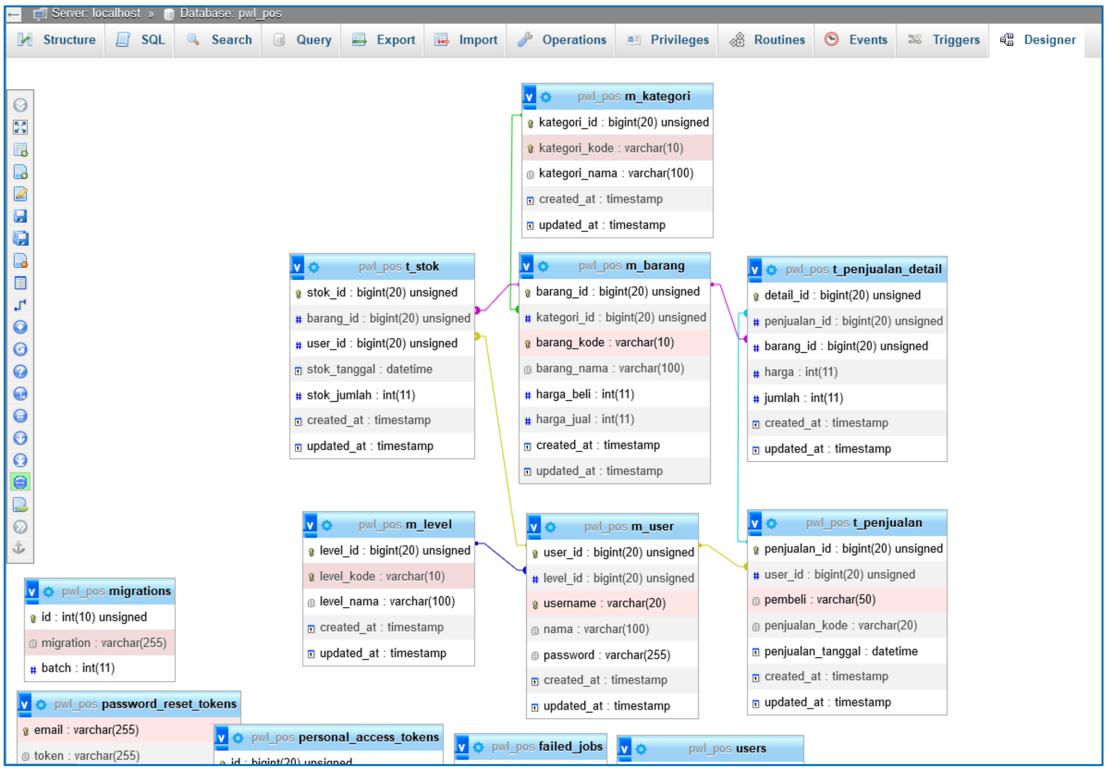
```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('m_user', function (Blueprint $table) {
            $table->id('user_id');
            $table->unsignedBigInteger('level_id')->index(); // indexing untuk ForeignKey
            $table->string('username', 20)->unique(); // unique untuk memastikan tidak ada username yang sama
            $table->string('nama', 100);
            $table->string('password');
            $table->timestamps();

            // Mendefinisikan Foreign Key pada kolom level_id mengacu pada kolom level_id di tabel m_level
            $table->foreign('level_id')->references('level_id')->on('m_level');
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('m_user');
    }
};
```

3. Save the program code Step 2, and run the `php artisan migrate command`. Observe what happens to the database.

4. Create *database* tables with *migration* for tables that have *foreign keys*

`m_barang`
`t_penjualan`
`t_stok`
`t_penjualan_detail`

5. If all migration files have been created and run, then we can see the ***`designer`*** display in **`phpMyAdmin`** as follows:



pwl_pos **m_kategori**
kategori_id : bigint(20) unsigned
kategori_kode : varchar(10)
kategori_nama : varchar(100)
created_at : timestamp
updated_at : timestamp

pwl_pos **t_stok**
stok_id : bigint(20) unsigned
barang_id : bigint(20) unsigned
user_id : bigint(20) unsigned
stok_tanggal : datetime
stok_jumlah : int(11)
created_at : timestamp
updated_at : timestamp

pwl_pos **m_barang**
barang_id : bigint(20) unsigned
kategori_id : bigint(20) unsigned
barang_kode : varchar(10)
barang_nama : varchar(100)
harga_beli : int(11)
harga_jual : int(11)
created_at : timestamp
updated_at : timestamp

pwl_pos **t_penjualan_detail**
detail_id : bigint(20) unsigned
penjualan_id : bigint(20) unsigned
barang_id : bigint(20) unsigned
harga : int(11)
jumlah : int(11)
created_at : timestamp
updated_at : timestamp

pwl_pos **m_level**
level_id : bigint(20) unsigned
level_kode : varchar(10)
level_nama : varchar(100)
created_at : timestamp
updated_at : timestamp

pwl_pos **m_user**
user_id : bigint(20) unsigned
level_id : bigint(20) unsigned
username : varchar(20)
nama : varchar(100)
password : varchar(255)
created_at : timestamp
updated_at : timestamp

pwl_pos **t_penjualan**
penjualan_id : bigint(20) unsigned
user_id : bigint(20) unsigned
pembeli : varchar(50)
penjualan_kode : varchar(20)
penjualan_tanggal : datetime
created_at : timestamp
updated_at : timestamp

pwl_pos **migrations**
id : int(10) unsigned
migration : varchar(255)
batch : int(11)

pwl_pos **password_reset_tokens**
email : varchar(255)
token : varchar(255)

pwl_pos **personal_access_tokens**
id : bigint(20) unsigned

pwl_pos **failed_jobs**

pwl_pos **users**

6. Report the results of this Practicum-2.2 and *commit* changes to *git*.

C. SEEDER

Seeder is a feature that allows us to populate our database with initial data or *predetermined dummy* data . Seeders allow us to create the same initial data for each use in application development. Generally, the data that is often created by *seeders* is user data because the data will be used when the application is first run and requires *login actions*.

1. The general command in **creating a** `seeder file` is as follows:

```
php artisan make:seeder <nama-class-seeder>
```

   The command will generate seeder files in the `PWL_POS/database/seeders folder`

2. And the command to **run the** `seeder file` is as follows

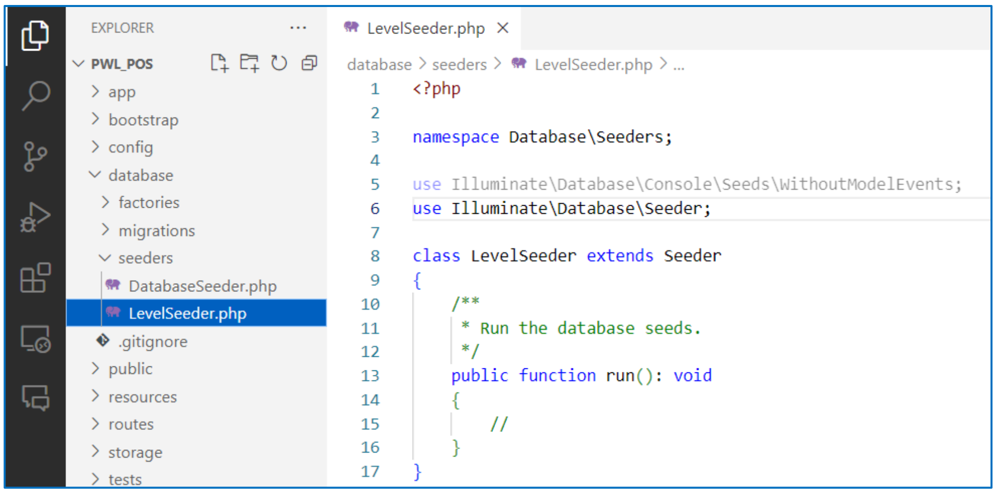
```
php artisan db:seed --class=<nama-class-seeder>
```

In the process of developing an application, we often need dummy initial data to facilitate testing and development of our application. So that we can use the *seeder* feature in making a web application.

Practicum 3 – Creating seeder files

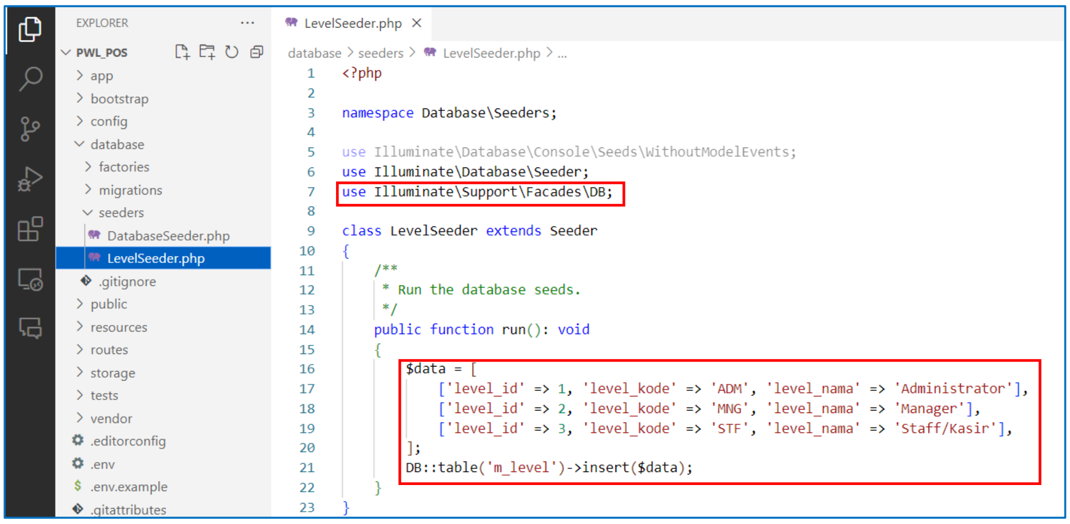
1. We will create a seeder file for the m_level table by typing the command

```
php artisan make:seeder LevelSeeder
```



```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class LevelSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        //
    }
}
```

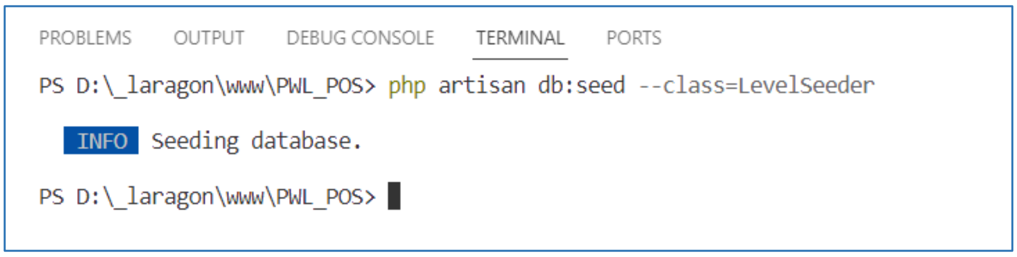
2. Next, to enter the initial data, we modify the file inside the `run() function`



```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class LevelSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $data = [
            ['level_id' => 1, 'level_kode' => 'ADM', 'level_nama' => 'Administrator'],
            ['level_id' => 2, 'level_kode' => 'MNG', 'level_nama' => 'Manager'],
            ['level_id' => 3, 'level_kode' => 'STF', 'level_nama' => 'Staff/Kasir'],
        ];
        DB::table('m_level')->insert($data);
    }
}
```

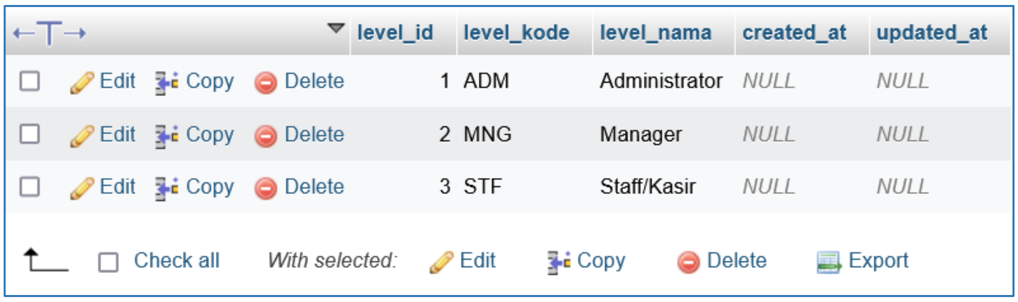
3. Next, we run the *seeder file* for the m_level table on the terminal

```
php artisan db:seed --class=LevelSeeder
```



```
PS D:\_laragon\www\PWL_POS> php artisan db:seed --class=LevelSeeder

   INFO  Seeding database.

PS D:\_laragon\www\PWL_POS> 
```

4. When the *seeder* is successfully run, data will appear on the table `m_level`

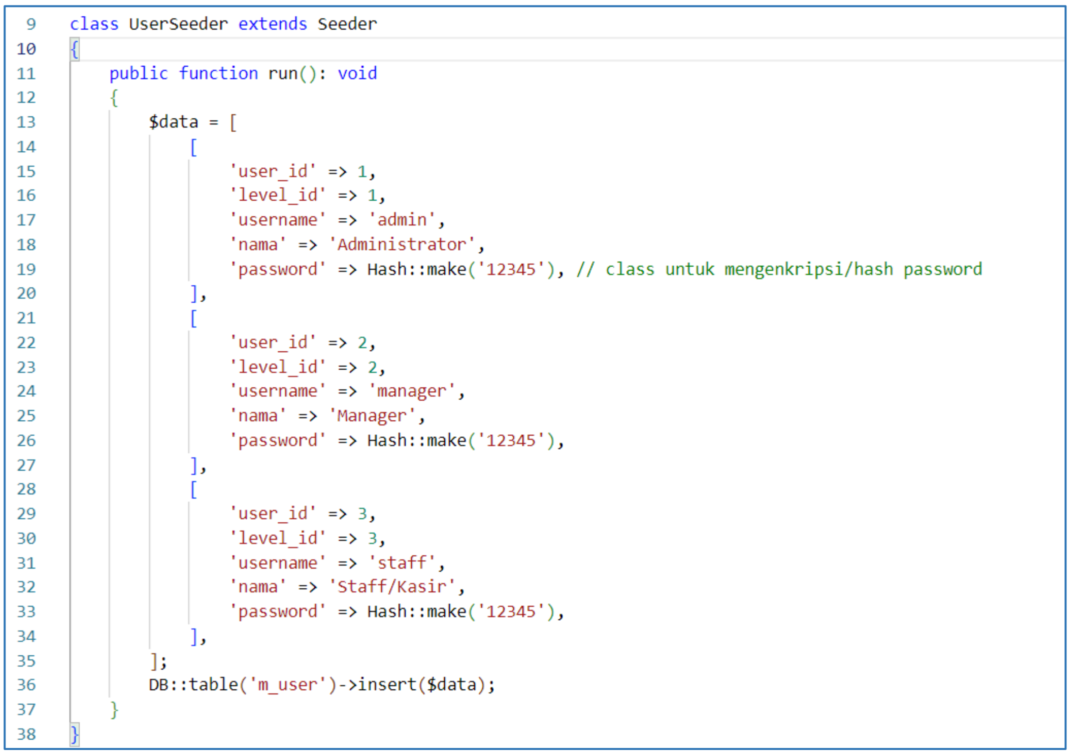


	level_id	level_kode	level_nama	created_at	updated_at
Edit Copy Delete	1	ADM	Administrator	NULL	NULL
Edit Copy Delete	2	MNG	Manager	NULL	NULL
Edit Copy Delete	3	STF	Staff/Kasir	NULL	NULL

5. Now we create a *seeder file* for the m_user table that refers to the table `m_level`

```
php artisan make:seeder UserSeeder
```

6. Modify the `UserSeeder class` file as follows

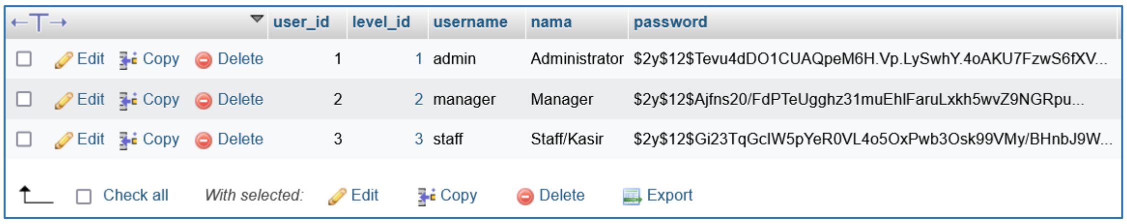


```php
class UserSeeder extends Seeder
{
    public function run(): void
    {
        $data = [
            [
                'user_id' => 1,
                'level_id' => 1,
                'username' => 'admin',
                'nama' => 'Administrator',
                'password' => Hash::make('12345'), // class untuk mengenkripsi/hash password
            ],
            [
                'user_id' => 2,
                'level_id' => 2,
                'username' => 'manager',
                'nama' => 'Manager',
                'password' => Hash::make('12345'),
            ],
            [
                'user_id' => 3,
                'level_id' => 3,
                'username' => 'staff',
                'nama' => 'Staff/Kasir',
                'password' => Hash::make('12345'),
            ],
        ];
        DB::table('m_user')->insert($data);
    }
}
```

7. Run the command to execute the `UserSeeder class`

```
php artisan db:seed --class=UserSeeder
```

8. Pay attention to the seeder results on the table `m_user`



	user_id	level_id	username	nama	password
☐ Edit Copy Delete	1	1	admin	Administrator	$2y$12$Tevu4dDO1CUAQpeM6H.Vp.LySwhY.4oAKU7FzwS6fXV...
☐ Edit Copy Delete	2	2	manager	Manager	$2y$12$Ajfns20/FdPTeUgghz31muEhlFaruLxkh5wvZ9NGRpu...
☐ Edit Copy Delete	3	3	staff	Staff/Kasir	$2y$12$Gi23TqGclW5pYeR0VL4o5OxPwb3Osk99VMy/BHnbJ9W...

9. Ok, the seeder data was successfully entered into the database.
10. Now try to enter the *seeder* data for another table, with conditions like the following

No	Table Name	Amount of Data	Information
1	`m_kategori`	5	5 categories of goods
2	`m_barang`	10	10 different items
3	`t_stok`	10	Stock for 10 items
4	`t_penjualan`	10	10 sales transactions
5	`t_penjualan_detail`	30	3 items for each sales transaction

11. If so, report the results of Practicum-3 and *commit* changes to *git*

D. DB FACADE

DB Façade is a feature of Laravel that is used to query directly by typing the SQL request as a whole (*raw query*). It is called *a raw* query because the query writing on the DB Façade is directly written as it is usually written in the database, such as `"select * from m_user"` or `"insert into m_user..."` or `"update m_user set... Where..."`

Raw queries are the most basic and traditional way in Laravel. Raw queries feel familiar because we usually use them when querying directly to the database.

INFO

Documentation of DB Façade usage can be checked on this page

`https://laravel.com/docs/10.x/database#running-queries`

There are many methods that can be used in this DB Façade. However, what we learned is quite 4 (four) commonly used methods, namely

a. `DB::select()`

This method is used to retrieve data from the database. This method **returns *the*** query result data. Example

```
DB::select('select * from m_user'); //Query semua data pada tabel m_user
```

```
DB::select('select * from m_user where level_id = ?', [1]); //Query tabel m_user dengan level_id = 1
```

```
DB::select('select * from m_user where level_id = ? and username = ?', [1, 'admin']);
```

b. `DB::insert()`

This method is used to enter data in the database table. This method **has no *return***. Example

```
DB::insert('insert into m_level(level_kode, level_nama) values(?,?)', ['CUS', 'Pelanggan']);
```

c. `DB::update()`

This method is used when running a *raw query* to update data in the database. This method **has a return value** in the form of the number of rows of data that are *updated*. Example

```
DB::update('update m_level set level_nama = ? where level_kode = ?', ['Customer', 'CUS']);
```

d. `DB::delete()`

This method is used when running a *raw query* to remove data from a table. This method **has a return value *in*** the form of the number of rows of data that have been deleted. Example

```
DB::delete('delete from m_level where level_kode = ?', ['CUS']);
```

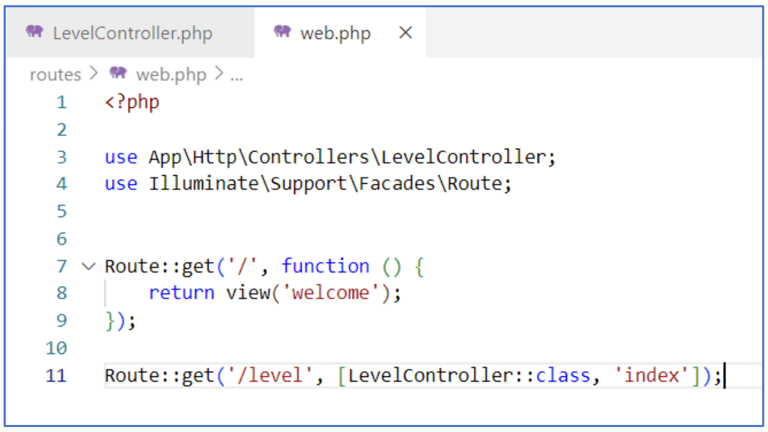
Ok, now let's try to practice using DB Façade in our project

Practicum 4 – DB Facade Implementation

1. We create a controller first to manage the data in the table `m_level`

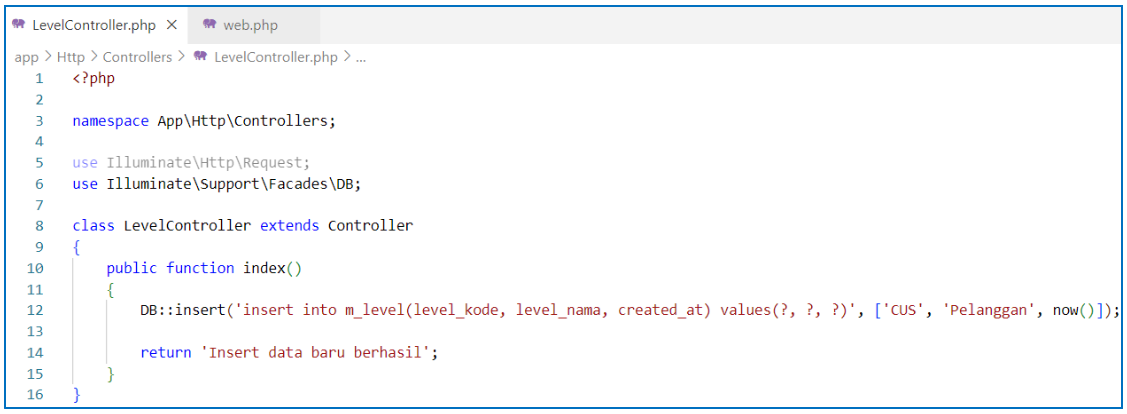
```
php artisan make:controller LevelController
```

2. We modify it first for the *routing*, it's in the `PWL_POS/routes/web.php`



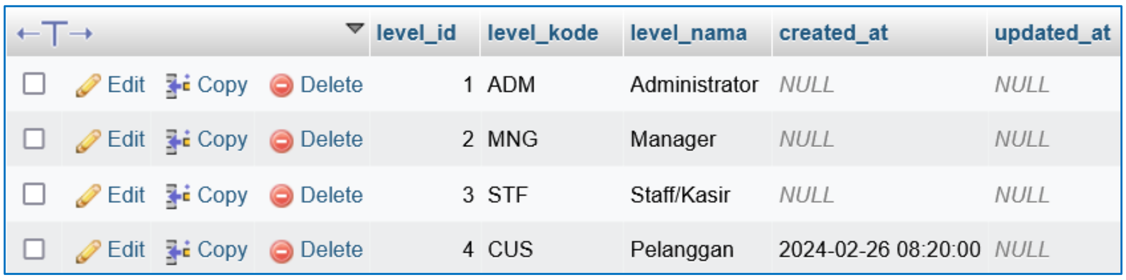
```php
<?php

use App\Http\Controllers\LevelController;
use Illuminate\Support\Facades\Route;


Route::get('/', function () {
    return view('welcome');
});

Route::get('/level', [LevelController::class, 'index']);
```

3. Next, we modify the `LevelController` file to add 1 data to the table `m_level`



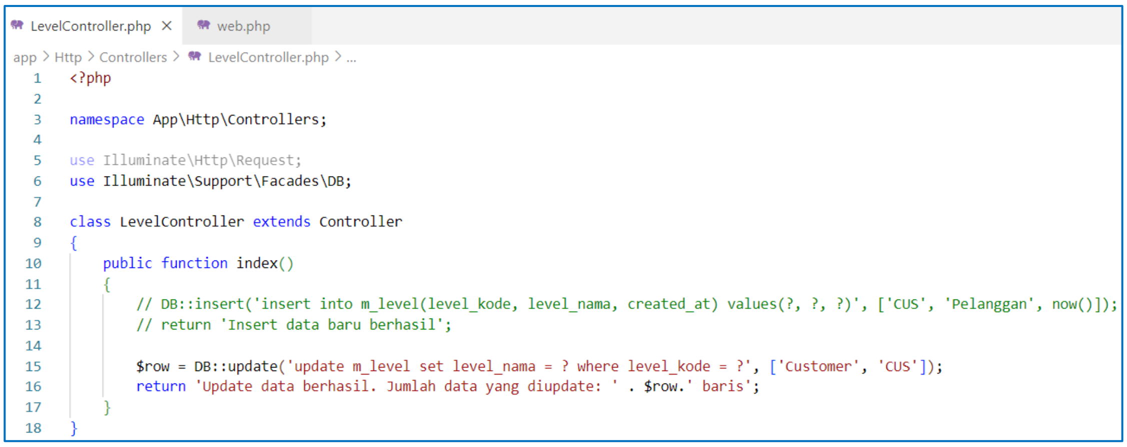
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class LevelController extends Controller
{
    public function index()
    {
        DB::insert('insert into m_level(level_kode, level_nama, created_at) values(?, ?, ?)', ['CUS', 'Pelanggan', now()]);

        return 'Insert data baru berhasil';
    }
}
```

4. We try to run it in a browser with the url `localhost/PWL_POS/public/level` and observe what happens to the table `m_level` in the database, *screenshot* the changes in the table `m_level`



	level_id	level_kode	level_nama	created_at	updated_at
Edit Copy Delete	1	ADM	Administrator	NULL	NULL
Edit Copy Delete	2	MNG	Manager	NULL	NULL
Edit Copy Delete	3	STF	Staff/Kasir	NULL	NULL
Edit Copy Delete	4	CUS	Pelanggan	2024-02-26 08:20:00	NULL

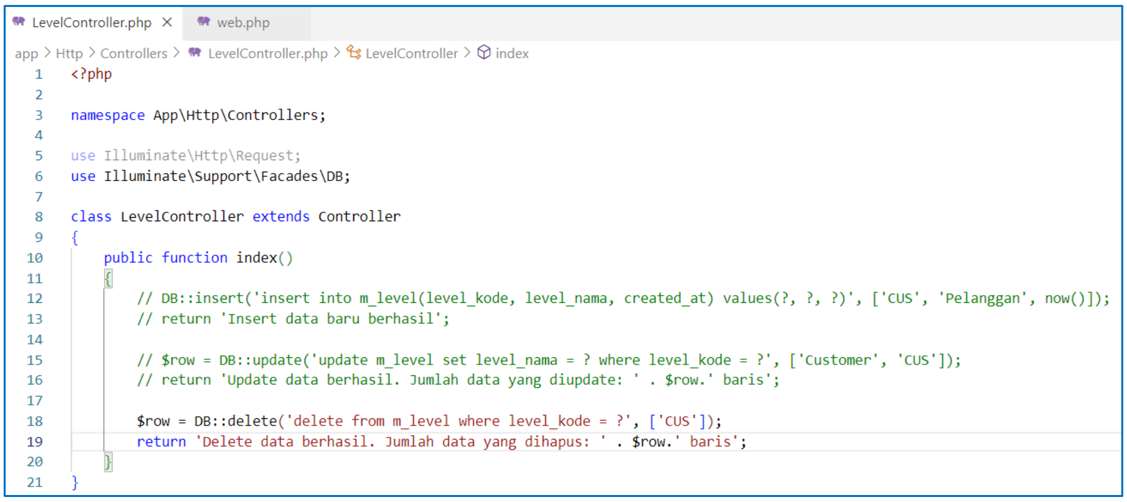
5. Next, we modify the `LevelController` file again to *update the* data in the table `m_level` as follows



LevelController.php × web.php
app > Http > Controllers > LevelController.php > ...

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class LevelController extends Controller
{
    public function index()
    {
        // DB::insert('insert into m_level(level_kode, level_nama, created_at) values(?, ?, ?)', ['CUS', 'Pelanggan', now()]);
        // return 'Insert data baru berhasil';

        $row = DB::update('update m_level set level_nama = ? where level_kode = ?', ['Customer', 'CUS']);
        return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';
    }
}
```

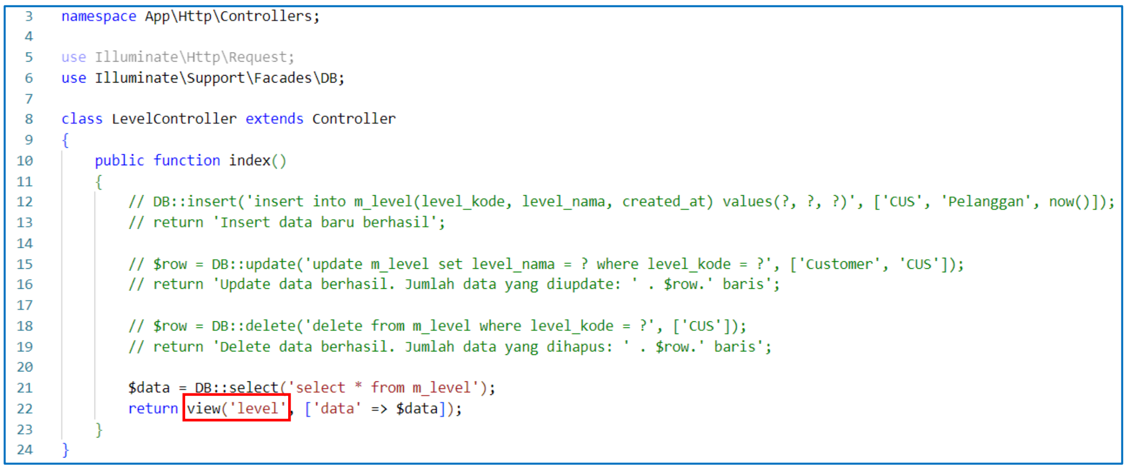
6. Let's try running it in the browser with the url `localhost/PWL_POS/public/level` again and observe what happens to the table `m_level` in the database, *screenshot* the changes in the table `m_level`
7. We try modifying the `LevelController` file again to delete the data



LevelController.php × web.php
app > Http > Controllers > LevelController.php > LevelController > index

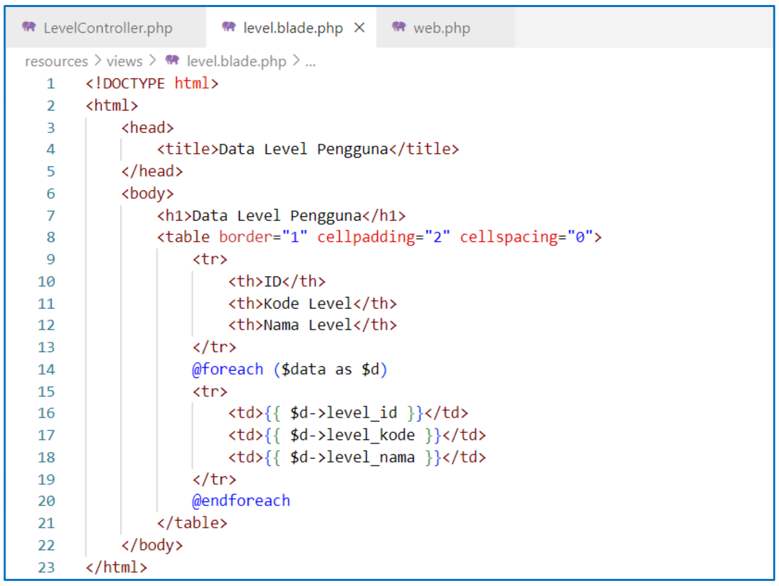
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class LevelController extends Controller
{
    public function index()
    {
        // DB::insert('insert into m_level(level_kode, level_nama, created_at) values(?, ?, ?)', ['CUS', 'Pelanggan', now()]);
        // return 'Insert data baru berhasil';

        // $row = DB::update('update m_level set level_nama = ? where level_kode = ?', ['Customer', 'CUS']);
        // return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';

        $row = DB::delete('delete from m_level where level_kode = ?', ['CUS']);
        return 'Delete data berhasil. Jumlah data yang dihapus: ' . $row.' baris';
    }
}
```

8. The last method we try is to display the data in the table `m_level`. We modify the `LevelController` file as follows



```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class LevelController extends Controller
{
    public function index()
    {
        // DB::insert('insert into m_level(level_kode, level_nama, created_at) values(?, ?, ?)', ['CUS', 'Pelanggan', now()]);
        // return 'Insert data baru berhasil';

        // $row = DB::update('update m_level set level_nama = ? where level_kode = ?', ['Customer', 'CUS']);
        // return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';

        // $row = DB::delete('delete from m_level where level_kode = ?', ['CUS']);
        // return 'Delete data berhasil. Jumlah data yang dihapus: ' . $row.' baris';

        $data = DB::select('select * from m_level');
        return view('level', ['data' => $data]);
    }
}
```

9. Let's look at the code marked with a red box, since the code calls `view('level')`, then we create a view file in VSCode at `PWL_POS/resources/view/level.blade.php`



LevelController.php | level.blade.php × | web.php

resources > views > level.blade.php > ...

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Data Level Pengguna</title>
    </head>
    <body>
        <h1>Data Level Pengguna</h1>
        <table border="1" cellpadding="2" cellspacing="0">
            <tr>
                <th>ID</th>
                <th>Kode Level</th>
                <th>Nama Level</th>
            </tr>
            @foreach ($data as $d)
            <tr>
                <td>{{ $d->level_id }}</td>
                <td>{{ $d->level_kode }}</td>
                <td>{{ $d->level_nama }}</td>
            </tr>
            @endforeach
        </table>
    </body>
</html>
```

10. Please try it in the browser and observe what happens
11. Report the results of this Practicum-4 and *commit* changes to *git*.

E. QUERY BUILDER

Query builder is a feature provided by Laravel to perform CRUD *(create, retrieve/read, update, delete)* processes on a database. Unlike *the raw query* on DB Facede which requires us to write SQL commands, in the *query builder* this SQL command is accessed using methods. So, we don't write SQL commands directly, but simply call the methods in the *query builder*.

Query builders make our code neat and easier to read. In addition, *the query builder* is not tied to one type of database, so it can be used to access various types of databases such as MySQL, MariaDB, PostgreSQL, SQL Server, etc. If one day you want to switch from MySQL to PostgreSQL database, there will not be many obstacles. But the disadvantage of the *query builder* is that we have to know what methods are in the *query builder*.

INFO

Documentation on using Query Builder on Laravel can be checked on this page

`https://laravel.com/docs/10.x/queries`

A distinctive feature of *the Laravel query builder* is that we first determine the target table that we will access for the CRUD operation.

```
DB::table('<nama-tabel>'); // query builder untuk melakukan operasi CRUD pada tabel yang dituju
```

The first command performed in the query builder is to specify the name of the table for which the CRUD operation will be performed. Then followed by the method that you want to use according to its designation. Example

a. Command to *insert* data with insert method `()`

```
DB::table('m_kategori')->insert(['kategori_kode' => 'SMP', 'kategori_nama' => 'Smartphone']);
```

The query generated from the above code is

```
insert into m_kategori(kategori_kode, kategori_nama) values('SMP', 'Smartphone');
```

b. Commands to *update* data with `where()` and `update() methods`

```
DB::table('m_kategori')->where('kategori_id', 1)->update(['kategori_nama' => 'Makanan Ringan']);
```

The query generated from the above code is

```
update m_kategori set kategori_nama='Snacks' where kategori_id=1;
```

c. Commands to *delete* data with `where()` and `delete() methods`

```
DB::table('m_kategori')->where('kategori_id', 9) ->delete();
```

The query generated from the above code is

```
delete from m_kategori where kategori_id = 9;
```

d. Command to retrieve data

Method Query Builder	**Query produced**
`DB::table('m_kategori')->get();`	`select * from m_kategori`

`DB::table('m_kategori')`
 `->where('kategori_id', 1)->get();`
	`select * from m_kategori where kategori_id = 1;`

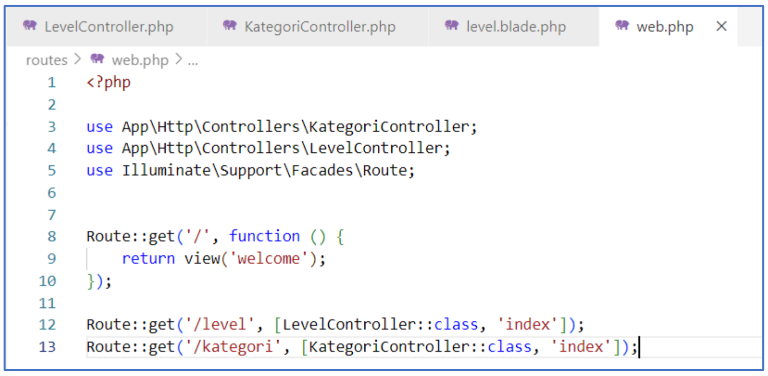
`DB::table('m_kategori')`
 `->select('kategori_kode')`
 `->where('kategori_id', 1)->get();`
	`select kategori_kode from m_kategori where kategori_id = 1;`

Practicum 5 – Query *Builder Implementation*

1. We create a controller to manage the data in the table `m_kategori`

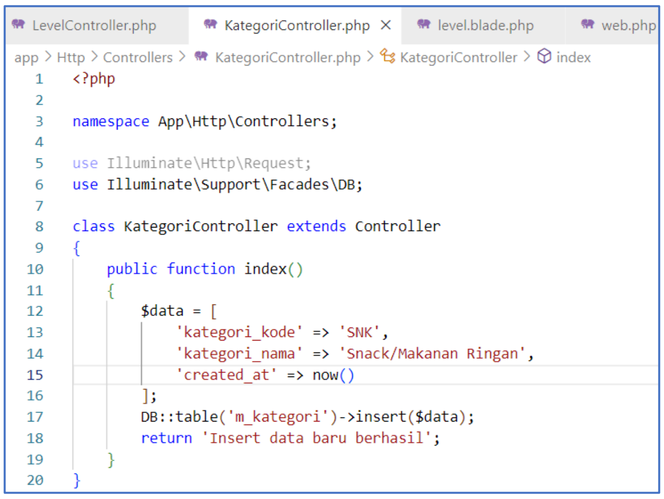
```
php artisan make:controller KategoriController
```

2. We modify it first for the routing, it's in the `PWL_POS/routes/web.php`



```php
<?php

use App\Http\Controllers\KategoriController;
use App\Http\Controllers\LevelController;
use Illuminate\Support\Facades\Route;


Route::get('/', function () {
    return view('welcome');
});

Route::get('/level', [LevelController::class, 'index']);
Route::get('/kategori', [KategoriController::class, 'index']);
```

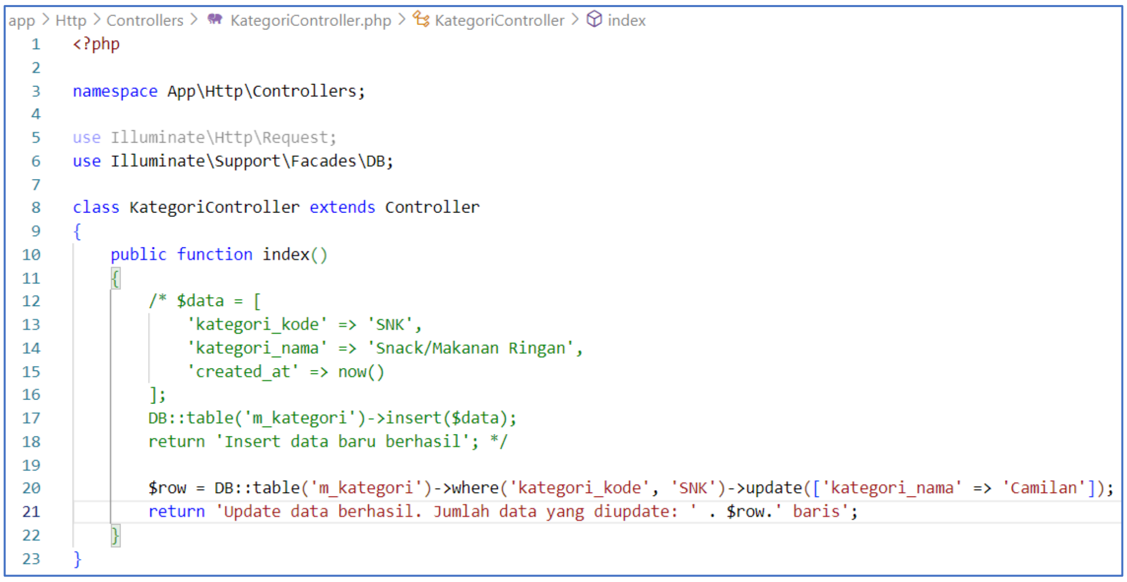
3. Next, we modify the `CategoryController` file to add 1 data to the table `m_kategori`



LevelController.php | KategoriController.php | level.blade.php | web.php
app > Http > Controllers > KategoriController.php > KategoriController > index

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class KategoriController extends Controller
{
    public function index()
    {
        $data = [
            'kategori_kode' => 'SNK',
            'kategori_nama' => 'Snack/Makanan Ringan',
            'created_at' => now()
        ];
        DB::table('m_kategori')->insert($data);
        return 'Insert data baru berhasil';
    }
}
```

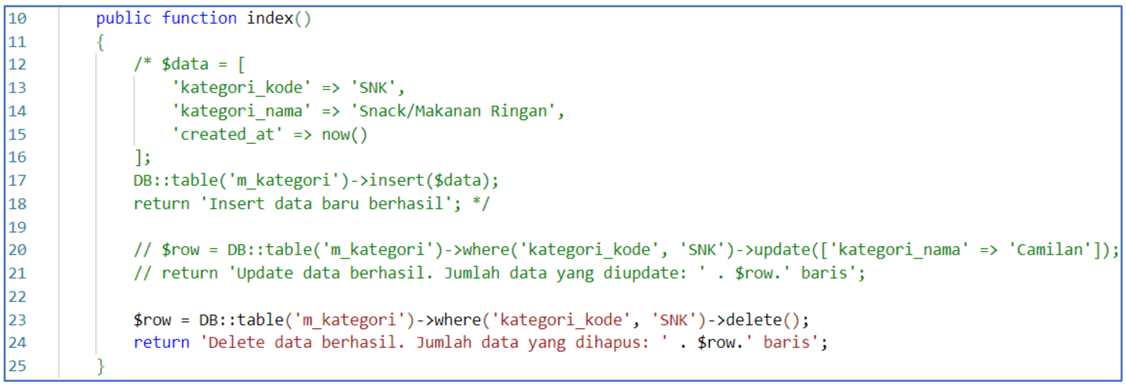
4. We try to run it in a browser with the url `localhost/PWL_POS/public/category` and observe what happens to the table `m_kategori` in the database, *screenshot* the changes in the table `m_kategori`
5. Next, we modify the `KategoriController` file again to *update the* data in the m_kategori table as follows



app > Http > Controllers > KategoriController.php > KategoriController > index

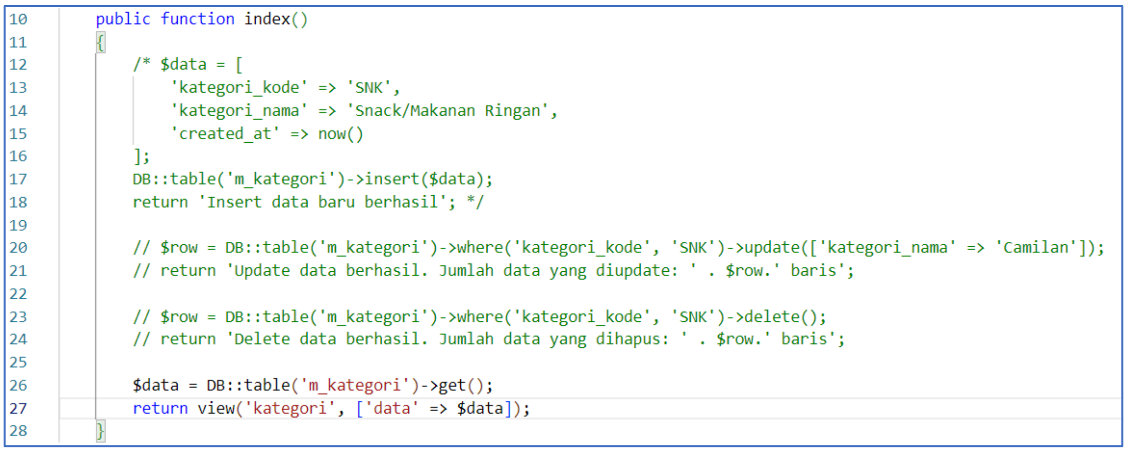
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class KategoriController extends Controller
{
    public function index()
    {
        /* $data = [
            'kategori_kode' => 'SNK',
            'kategori_nama' => 'Snack/Makanan Ringan',
            'created_at' => now()
        ];
        DB::table('m_kategori')->insert($data);
        return 'Insert data baru berhasil'; */

        $row = DB::table('m_kategori')->where('kategori_kode', 'SNK')->update(['kategori_nama' => 'Camilan']);
        return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';
    }
}
```

6. We try running it in the browser with the url `localhost/PWL_POS/public/category` again and observe what happens to the table `m_kategori` in the database, *screenshot* the changes in the table `m_kategori`
7. We try modifying the `KategoriController` file again to delete the data



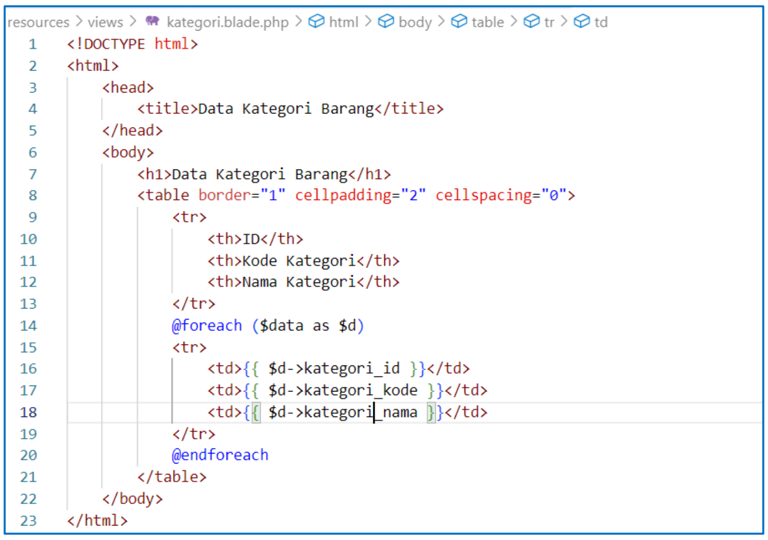
```
    public function index()
    {
        /* $data = [
            'kategori_kode' => 'SNK',
            'kategori_nama' => 'Snack/Makanan Ringan',
            'created_at' => now()
        ];
        DB::table('m_kategori')->insert($data);
        return 'Insert data baru berhasil'; */

        // $row = DB::table('m_kategori')->where('kategori_kode', 'SNK')->update(['kategori_nama' => 'Camilan']);
        // return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';

        $row = DB::table('m_kategori')->where('kategori_kode', 'SNK')->delete();
        return 'Delete data berhasil. Jumlah data yang dihapus: ' . $row.' baris';
    }
```

8. The last method we try is to display the data in the table `m_kategori`. We modify the `CategoryController` file as follows



```
    public function index()
    {
        /* $data = [
            'kategori_kode' => 'SNK',
            'kategori_nama' => 'Snack/Makanan Ringan',
            'created_at' => now()
        ];
        DB::table('m_kategori')->insert($data);
        return 'Insert data baru berhasil'; */

        // $row = DB::table('m_kategori')->where('kategori_kode', 'SNK')->update(['kategori_nama' => 'Camilan']);
        // return 'Update data berhasil. Jumlah data yang diupdate: ' . $row.' baris';

        // $row = DB::table('m_kategori')->where('kategori_kode', 'SNK')->delete();
        // return 'Delete data berhasil. Jumlah data yang dihapus: ' . $row.' baris';

        $data = DB::table('m_kategori')->get();
        return view('kategori', ['data' => $data]);
    }
```

9. Let's look at the code marked with a red box, since the code calls `view('category')`, then we create a view file in VSCode at `PWL_POS/resources/view/kategori.blade.php`



resources > views > kategori.blade.php > html > body > table > tr > td

```
<!DOCTYPE html>
<html>
    <head>
        <title>Data Kategori Barang</title>
    </head>
    <body>
        <h1>Data Kategori Barang</h1>
        <table border="1" cellpadding="2" cellspacing="0">
            <tr>
                <th>ID</th>
                <th>Kode Kategori</th>
                <th>Nama Kategori</th>
            </tr>
            @foreach ($data as $d)
            <tr>
                <td>{{ $d->kategori_id }}</td>
                <td>{{ $d->kategori_kode }}</td>
                <td>{{ $d->kategori_nama }}</td>
            </tr>
            @endforeach
        </table>
    </body>
</html>
```

10. Please try it in the browser and observe what happens.
11. Report the results of this Practicum-5 and *commit* changes to *git*

F. ELOQUENT ORM

Eloquent ORM is a built-in feature of laravel. Eloquent ORM is a way of accessing a database where each table row is considered an object. The word ORM itself stands for ***Object-relational mapping***, which is a programming technique to convert data into objects.

INFO

Eloquent ORM requires a Model for the process of converting data in tables into objects. This object is what we will later access from within the controller. Therefore **creating a Model on Laravel means using Eloquent ORM**. Please check here

`https://laravel.com/docs/10.x/eloquent`

The command to create a model is as follows

```
php artisan make:model <nama-model-CamelCase>
```

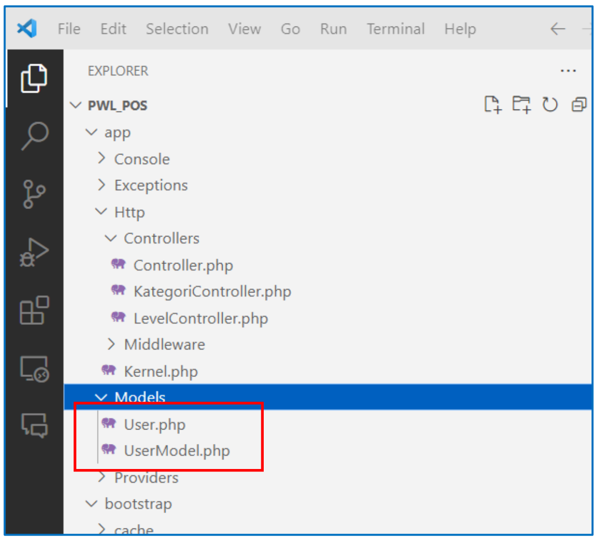
To be able to perform ***CRUD*** operations (*create, read/retrieve, update, delete*), we must create a model according to the target table we want to use. So

In 1 model, represents 1 database table.

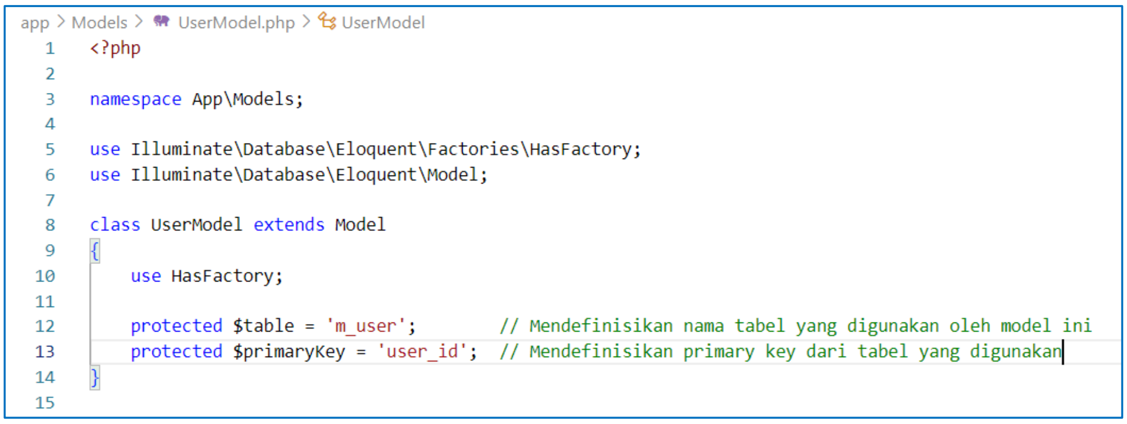
Practicum 6 – Implementation of Eloquent ORM

1. We create a model file for `the m_user` table by typing the command

```
php artisan make:model UserModel
```



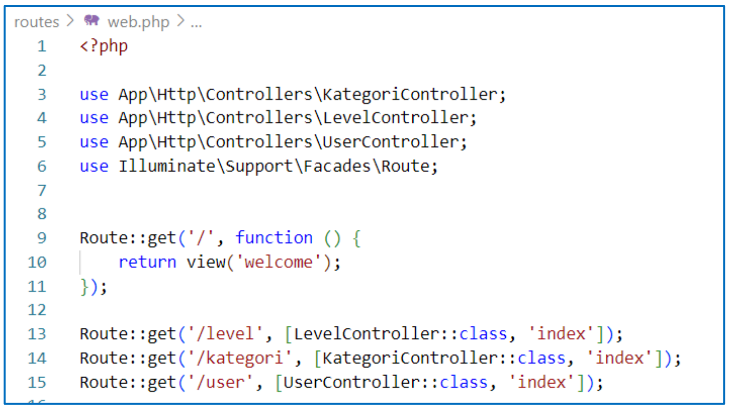
2. After successfully generating the model, there are 2 files in the `model` folder , namely `the default User.php` file from laravel and the `UserModel.php file` that we have created. This time we will use the UserModel.php
3. We open `the UserModel.php` file and modify it as follows



app > Models > UserModel.php > UserModel

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class UserModel extends Model
{
    use HasFactory;

    protected $table = 'm_user';        // Mendefinisikan nama tabel yang digunakan oleh model ini
    protected $primaryKey = 'user_id';  // Mendefinisikan primary key dari tabel yang digunakan
}

```

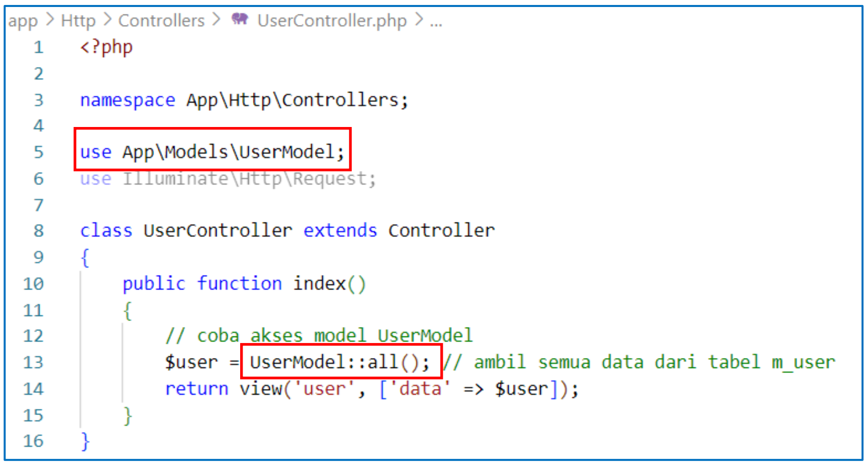
4. We modify the web.php route to try routing to the `UserController controller`



routes > web.php > ...

```php
<?php

use App\Http\Controllers\KategoriController;
use App\Http\Controllers\LevelController;
use App\Http\Controllers\UserController;
use Illuminate\Support\Facades\Route;


Route::get('/', function () {
    return view('welcome');
});

Route::get('/level', [LevelController::class, 'index']);
Route::get('/kategori', [KategoriController::class, 'index']);
Route::get('/user', [UserController::class, 'index']);
```

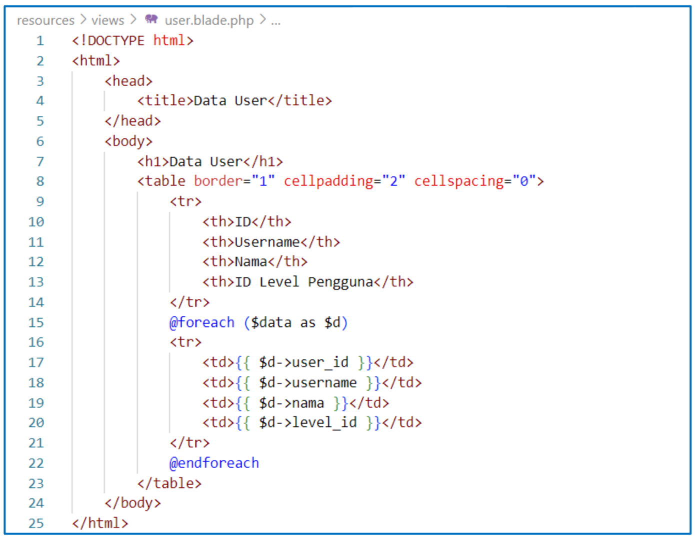
5. Now, we create a `UserController` controller file and modify it as follows



app > Http > Controllers > UserController.php > ...

```php
<?php

namespace App\Http\Controllers;

use App\Models\UserModel;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function index()
    {
        // coba akses model UserModel
        $user = UserModel::all(); // ambil semua data dari tabel m_user
        return view('user', ['data' => $user]);
    }
}
```

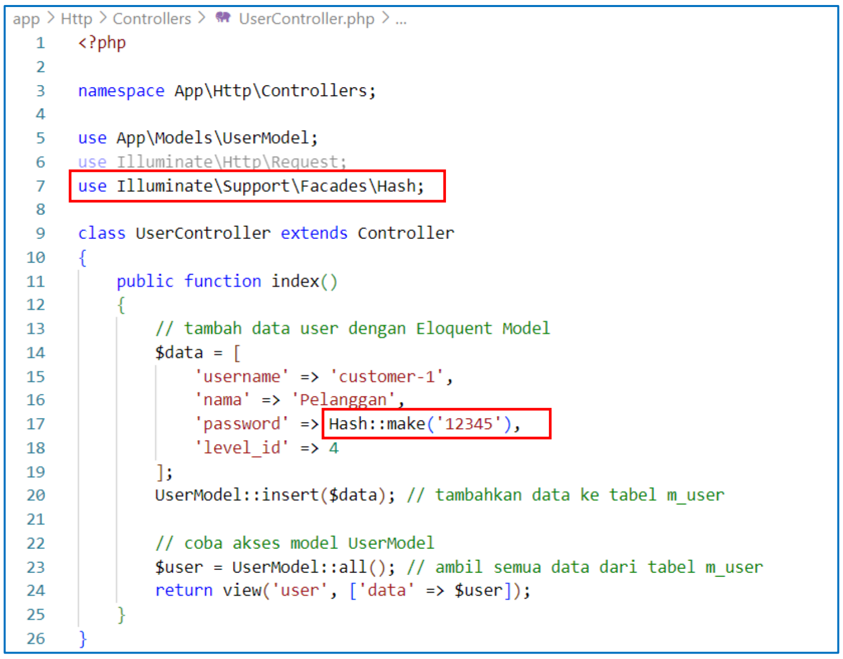
6. Then we create a view `user.blade.php`



resources > views > user.blade.php > ...

```
<!DOCTYPE html>
<html>
    <head>
        <title>Data User</title>
    </head>
    <body>
        <h1>Data User</h1>
        <table border="1" cellpadding="2" cellspacing="0">
            <tr>
                <th>ID</th>
                <th>Username</th>
                <th>Nama</th>
                <th>ID Level Pengguna</th>
            </tr>
            @foreach ($data as $d)
            <tr>
                <td>{{ $d->user_id }}</td>
                <td>{{ $d->username }}</td>
                <td>{{ $d->nama }}</td>
                <td>{{ $d->level_id }}</td>
            </tr>
            @endforeach
        </table>
    </body>
</html>
```

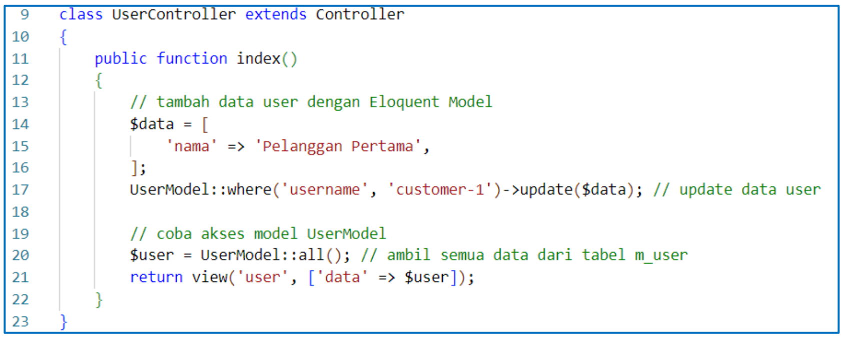
7. Run it in the browser, log and report what happened
8. After that, we modify the `UserController file again`



app > Http > Controllers > UserController.php > ...

```
<?php

namespace App\Http\Controllers;

use App\Models\UserModel;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class UserController extends Controller
{
    public function index()
    {
        // tambah data user dengan Eloquent Model
        $data = [
            'username' => 'customer-1',
            'nama' => 'Pelanggan',
            'password' => Hash::make('12345'),
            'level_id' => 4
        ];
        UserModel::insert($data); // tambahkan data ke tabel m_user

        // coba akses model UserModel
        $user = UserModel::all(); // ambil semua data dari tabel m_user
        return view('user', ['data' => $user]);
    }
}
```

9. Run it in the browser, observe and report what happened
10. We modify the `UserController file` again to look like this:



```
class UserController extends Controller
{
    public function index()
    {
        // tambah data user dengan Eloquent Model
        $data = [
            'nama' => 'Pelanggan Pertama',
        ];
        UserModel::where('username', 'customer-1')->update($data); // update data user

        // coba akses model UserModel
        $user = UserModel::all(); // ambil semua data dari tabel m_user
        return view('user', ['data' => $user]);
    }
}
```

11. Run it in the browser, observe and report what happened
12. If so, report the results of Practicum-6 and *commit* changes to *git*

G. Questions

Answer the following questions according to the understanding of the material above

1. In **Practicum 1 - Step 5**, what is the function of the `APP_KEY` in the *Laravel* `.env` setting file?
2. In **Practicum 1**, how do we *generate* value for `APP_KEY`?
3. In **Practicum 2.1 - Step 1**, *by default* how many migration files does Laravel have? and what are the migration files for?
4. *By default*, the migration file contains the code `$table->timestamps();`, what is the purpose/*output* of the function?
5. In the Migration File, there is a function `$table->id();` What type of data does the function return?
6. What is the difference between the migration results in the m_level table, between using `$table->id();` by using `$table->id('level_id');` ?
7. In migrations, what is `the ->unique()` function used for?
8. In **Practicum 2.2 - Step 2**, why does the `level_id column` in the m_user table use `$tabel->unsignedBigInteger('level_id')`, while the `level_id column` in the m_level table uses `$tabel->id('level_id')` ?
9. In **Practicum 3 - Step 6**, what is the purpose of the Hash Class? and what does the Hash program code mean`::make('1234');`?
10. In **Practicum 4 - Step 3/5/7**, in the *query builder* there is a question mark (`?`), what is the use of the question mark (`?`) of these?
11. In **Practicum 6 - Step 3**, what is the purpose of writing protected code `$table = 'm_user';` and `protected $primaryKey = 'user_id';` ?
12. In your opinion, where is it easier to use in performing CRUD operations to the database (*DB Façade / Query Builder / Eloquent ORM*) ?

*** Thank you, and good luck ***