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

JOBSHEET 04
MODEL and ELOQUENT ORM

Previously we discussed *about Migration, Seeder*, *DB Façade, Query Builder,* and *Eloquent ORM* in Laravel. Before we make 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 the initial data that we use before creating an application, such as administrator user data, system settings data, etc.

In this meeting, we will understand how to display data, change data, and delete data using the Eloquent technique.

In accordance with `Case Study PWL.pdf`, our Laravel 10 project is using **the PWL_POS repository.**
We will use PWL_POS project until the 12th meeting later as a project that we will study.

ORM (Object Relation Mapping) is a technique that converts a table into an object that will be easy to use. The object created has the same properties as the fields in the table. The ORM serves as a liaison and at the same time makes it easier for us to create applications that use relational databases to make our tasks more efficient.

Advantages of ORM

1. There are many features such as transactions, connection pooling, migrations, seeds, streams, and so on.
2. The query command has better performance, than we write it manually.
3. We write data models in only one place, making it easier to update, maintain, and reuse the code.
4. Allows us to make good use of OOP (object-oriented programming)

In Laravel itself, Eloquent ORM has been provided to make it easier for us to perform various queries to the database, and make our work easier because there is no need to write long sql queries to process data.

A. PROPERTY `$fillable` and `$guarded`

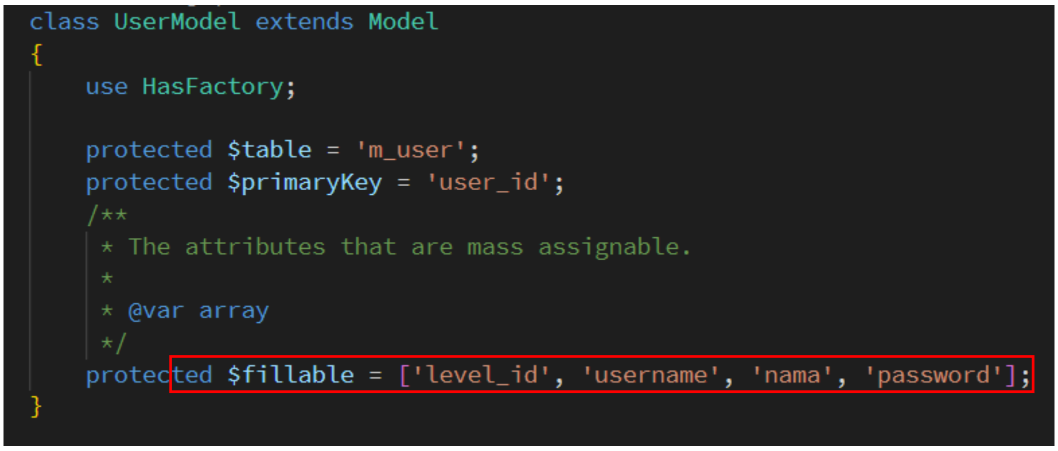
1. `$fillable`

`$fillable` variable useful for registering attributes (column names) that we can fill in when inserting or updating to the database. Earlier we already understood adding new records to the database. For the step of adding a variable `$fillable` you can add a *script* as below in the model file

```
protected $fillable = ['level_id', 'username'];
```

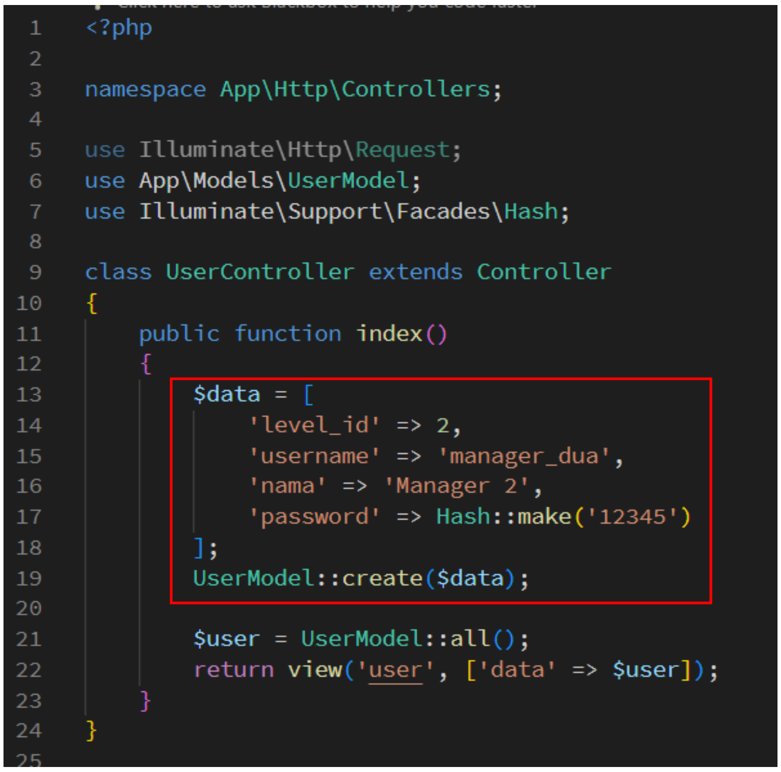
Practicum 1 - $fillable

1. Open the model file with the name `UserModel.php` and add $fillable as below screenshot shown.



```
class UserModel extends Model
{
    use HasFactory;

    protected $table = 'm_user';
    protected $primaryKey = 'user_id';
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['level_id', 'username', 'nama', 'password'];
}
```

2. Open the controller file with the name `UserController.php` and change the *script* to add new data as shown below.

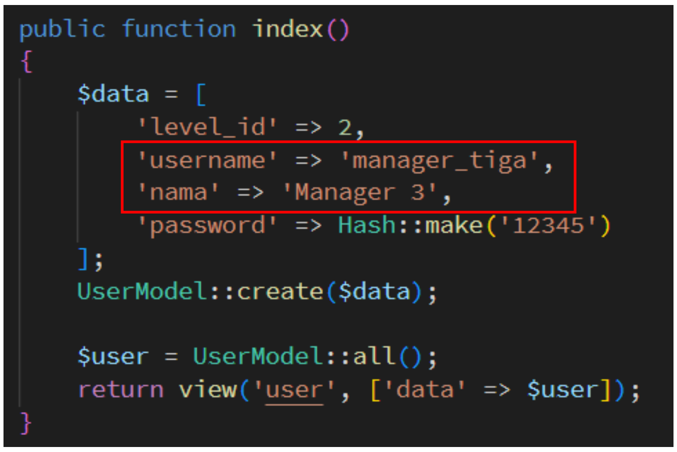


```php
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    public function index()
    {
        $data = [
            'level_id' => 2,
            'username' => 'manager_dua',
            'nama' => 'Manager 2',
            'password' => Hash::make('12345')
        ];
        UserModel::create($data);

        $user = UserModel::all();
        return view('user', ['data' => $user]);
    }
}
```

3. Save the program code of Steps 1 and 2, and run the web server commands. Then run the `localhostPWL_POS/public/user` in the *browser* and observe what happens.
4. Change the UserModel.php model file as shown below in the $fillable `section`

```php
protected $fillable = ['level_id', 'username', 'nama'];
```

5. Change the file controller `UserController.php` as shown below only the array section on the *$data*.



```php
public function index()
{
    $data = [
        'level_id' => 2,
        'username' => 'manager_tiga',
        'nama' => 'Manager 3',
        'password' => Hash::make('12345')
    ];
    UserModel::create($data);

    $user = UserModel::all();
    return view('user', ['data' => $user]);
}
```

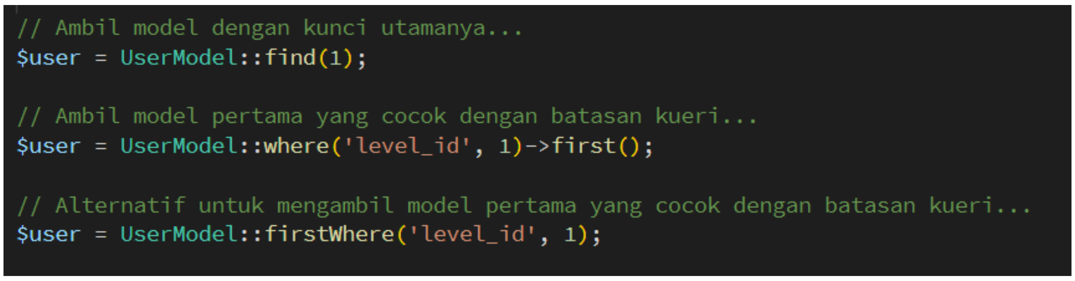
6. Save the program code of Steps 4 and 5. Then run it on *the browser* and observe what happens.
7. Report the results of this Practicum-1 and *commit* changes to *git*.

2. `$guarded`

The opposite of `$fillable` is `$guarded`. All columns we add to `$guarded` will be ignored by Eloquent when we insert/update. By default `$guarded` the contents are `array("*")`, which means all attributes cannot be set via ***mass assignment***. ***Mass Assignment*** is a powerful feature that simplifies the process of setting up multiple model attributes at once, saving time and effort. In this practicum, we'll explore the concept of mass assignment in Laravel and how it can be effectively leveraged to improve your development workflow.

B. RETRIEVING SINGLE MODELS

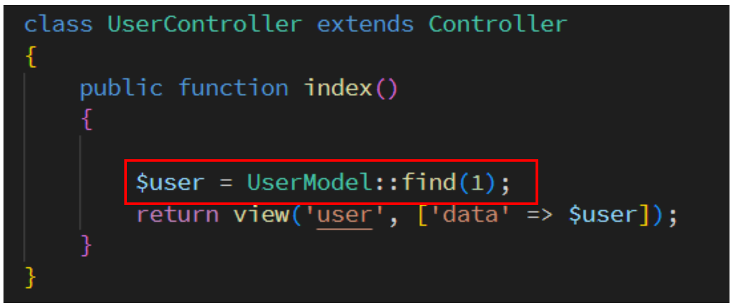
In addition to retrieving all records that match a particular query, you can also retrieve a single record using the `find`, `first`, or `firstWhere methods`. Rather than returning a collection of models, this method returns a single model instance and is performed on the controller:



```php
// Ambil model dengan kunci utamanya...
$user = UserModel::find(1);

// Ambil model pertama yang cocok dengan batasan kueri...
$user = UserModel::where('level_id', 1)->first();

// Alternatif untuk mengambil model pertama yang cocok dengan batasan kueri...
$user = UserModel::firstWhere('level_id', 1);
```

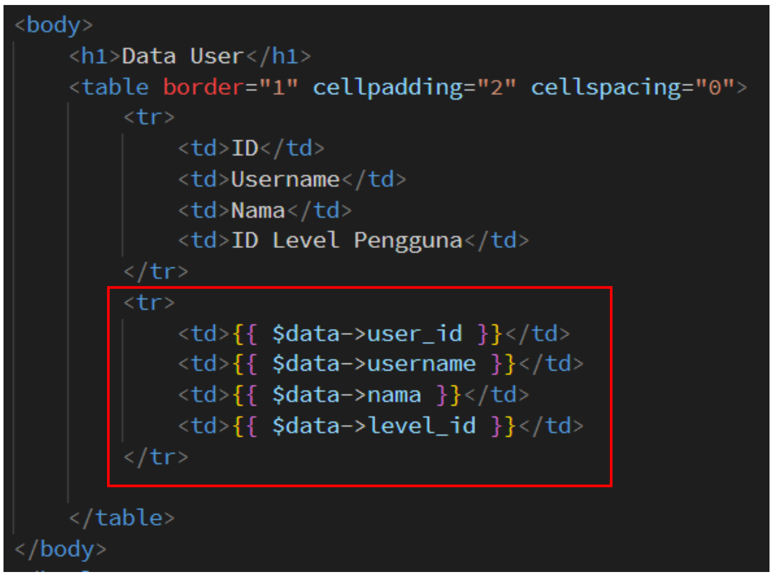
Praktikum 2.1 – Retrieving Single Models

1. Open the controller file `UserController.php` and change the *script* as shown below.



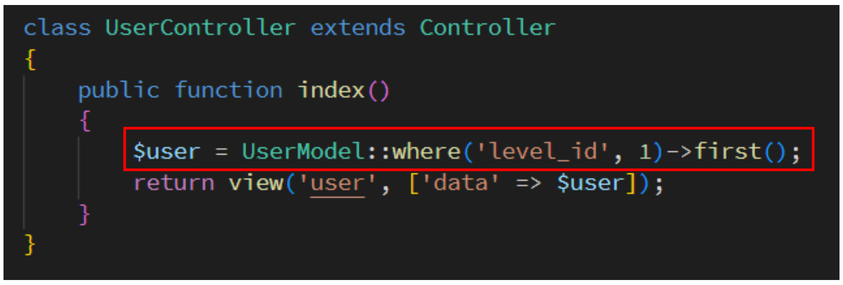
```php
class UserController extends Controller
{
    public function index()
    {

        $user = UserModel::find(1);
        return view('user', ['data' => $user]);
    }
}
```

2. Open the *view file* with the name `user.blade.php` and change the *script* as below screenshot shown



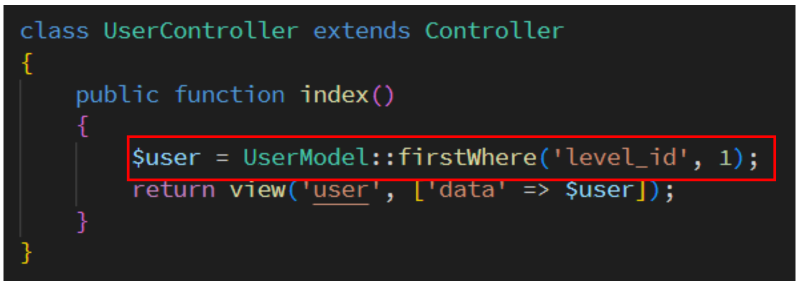
```
<body>
    <h1>Data User</h1>
    <table border="1" cellpadding="2" cellspacing="0">
        <tr>
            <td>ID</td>
            <td>Username</td>
            <td>Nama</td>
            <td>ID Level Pengguna</td>
        </tr>
        <tr>
            <td>{{ $data->user_id }}</td>
            <td>{{ $data->username }}</td>
            <td>{{ $data->nama }}</td>
            <td>{{ $data->level_id }}</td>
        </tr>

    </table>
</body>
```

3. Save the program code of Steps 1 and 2. Then run it in the *browser* and observe what happens and give an explanation in the report
4. Change the controller file with the name `UserController.php` and change the *script* as shown below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::where('level_id', 1)->first();
        return view('user', ['data' => $user]);
    }
}
```

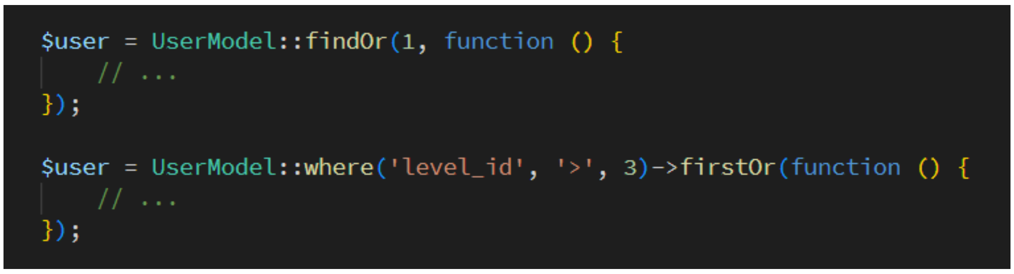
5. Save the program code Step 4. Then run it in the *browser* and observe what happens and give an explanation in the report
6. Change the controller file with the name `UserController.php` and change the *script* as shown below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstWhere('level_id', 1);
        return view('user', ['data' => $user]);
    }
}
```

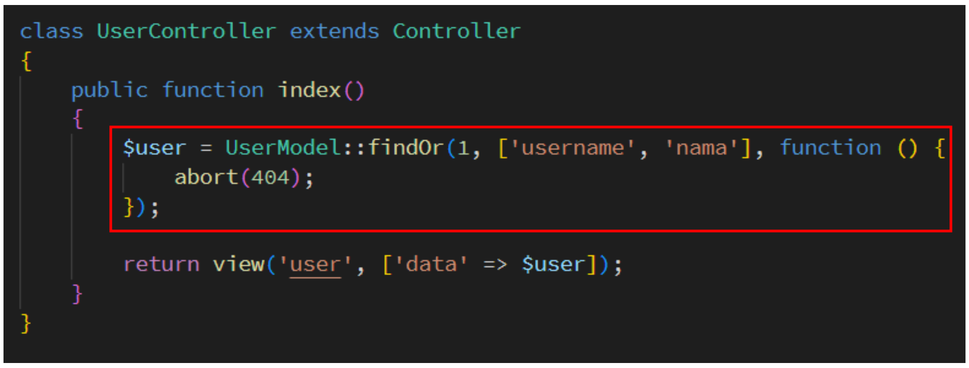
7. Save the program code Step 6. Then run it in the *browser* and observe what happens and give an explanation in the report

Sometimes you may want to perform some other actions if no results are found. The `findOr` and `firstOr` methods will return a single model instance or, if no results are found, will run inside the function. The value returned by the function will be considered as the result of this method:



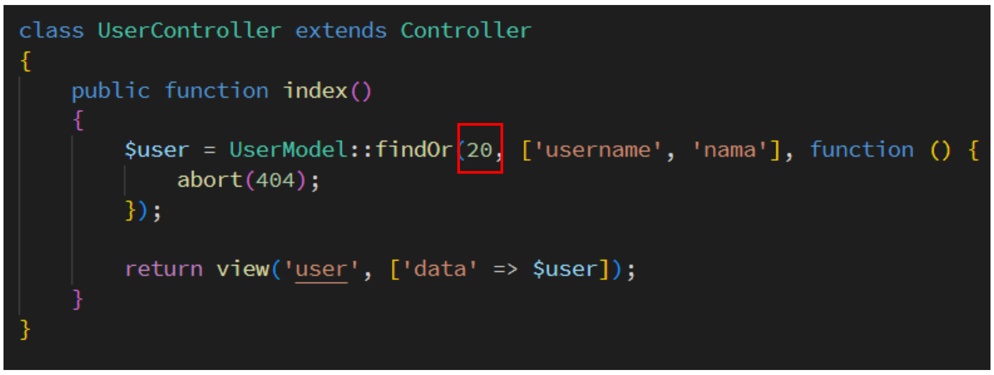
```php
$user = UserModel::findOr(1, function () {
    // ...
});

$user = UserModel::where('level_id', '>', 3)->firstOr(function () {
    // ...
});
```

8. Change the controller file with the name `UserController.php` and change the *script* as shown below



```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::findOr(1, ['username', 'nama'], function () {
            abort(404);
        });

        return view('user', ['data' => $user]);
    }
}
```

9. Save the program code Step 8. Then open it in the *browser* and observe what is happening and give an explanation in the report
10. Change the controller file with the name `UserController.php` and change the *script* as shown below



```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::findOr(20, ['username', 'nama'], function () {
            abort(404);
        });

        return view('user', ['data' => $user]);
    }
}
```

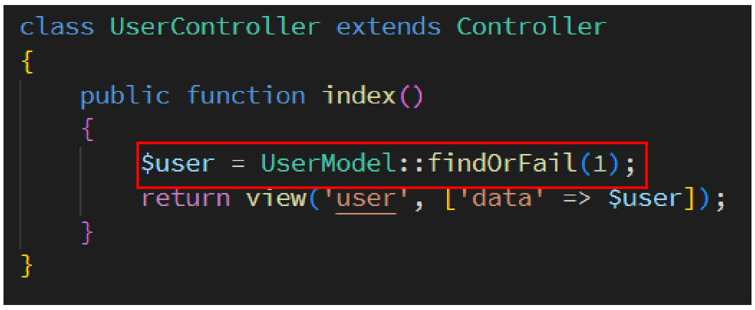
11. Save the program code Step 10. Then run it in the *browser* and observe what happens and give an explanation in the report.

12. Report the results of this Practicum-2.1 and *commit* changes to *git*.

Praktikum 2.2 – *Not Found Exceptions*

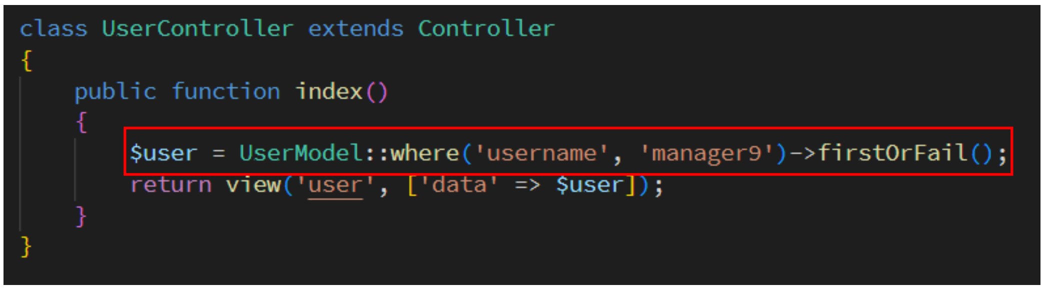
Sometimes you may want to throw an exception if the model is not found. This is especially useful in *routes* or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no results are found, an `Illuminate\Database\Eloquent\ModelNotFoundException` is thrown. Here follow the steps below:

1. Change the controller file with the name `UserController.php` and change the *script* as shown below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::findOrFail(1);
        return view('user', ['data' => $user]);
    }
}
```

2. Save the program code Step 1. Then run it in the *browser* and observe what happens and give an explanation in the report
3. Change the controller file with the name `UserController.php` and change the *script* as shown below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::where('username', 'manager9')->firstOrFail();
        return view('user', ['data' => $user]);
    }
}
```

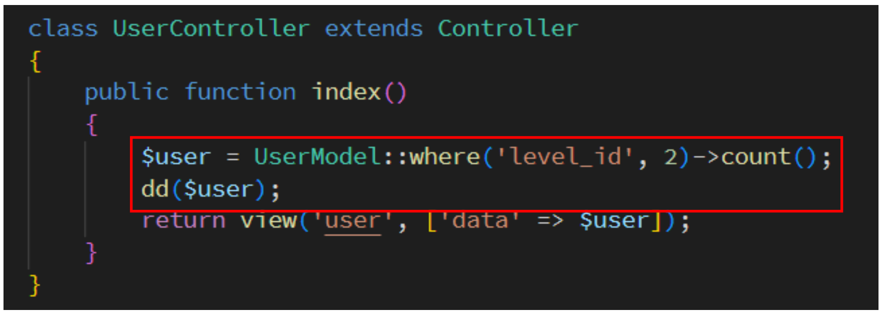
4. Save the program code Step 3. Then run it in the *browser* and observe what happens and give an explanation in the report
5. Report the results of this Practicum-2.2 and *commit* changes to *git*.

Lab Course 2.3 – *Retreiving Aggregrates*

When interacting with Eloquent models, you can also use the aggregate methods `count`, `sum`, `max`, and more provided by the Laravel query builder. As you might expect, this method returns scalar values and an example of the Eloquent model:

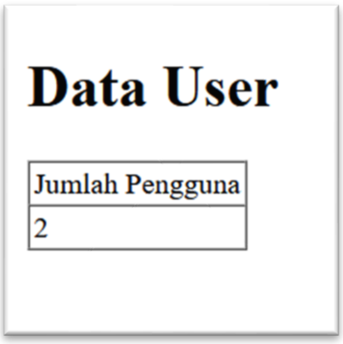
```
$count = UserModel::where('active', 1)->count();

$max = UserModel::where('active', 1)->max('price');
```

1. Change the controller file with the name `UserController.php` and change the *script* as shown below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::where('level_id', 2)->count();
        dd($user);
        return view('user', ['data' => $user]);
    }
}
```

2. Save the program code Step 1. Then run it in the *browser* and observe what happens and give an explanation in the report
3. Make it so that the number of *scripts* in step 1 can appear on the *browser page*, for example you can see the image below and change the *script* in the view file so that the data can appear



Data User

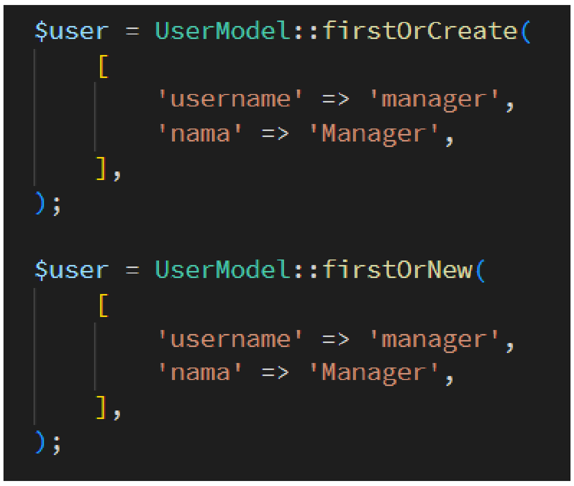
Jumlah Pengguna
2

4. Report the results of this Practicum-2.3 and *commit* changes to *git*.

Praktikum 2.4 – *Retreiving or Creating Models*

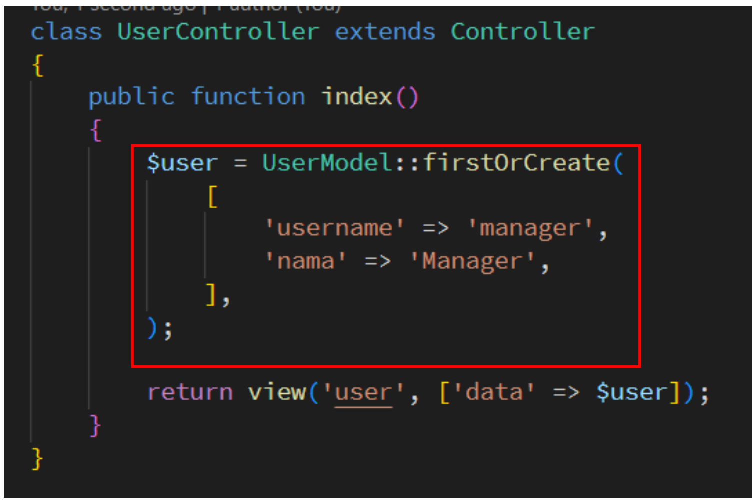
The `firstOrCreate` method is a method to *retrieving data* based on the value you want to search, if the data is not found then this method will insert it into the datadase table according to the value entered.

The `firstOrNew` `method`, such as `firstOrCreate`, will attempt to find/retrieve *records/data* in the database that match the given attribute. However, if no data is found, it will be prepared for *insertion* into the database and a new model will be returned. Note that the model returned `by` `firstOrNew` has not been saved to the database. You need to manually call the `save()` method to save it:



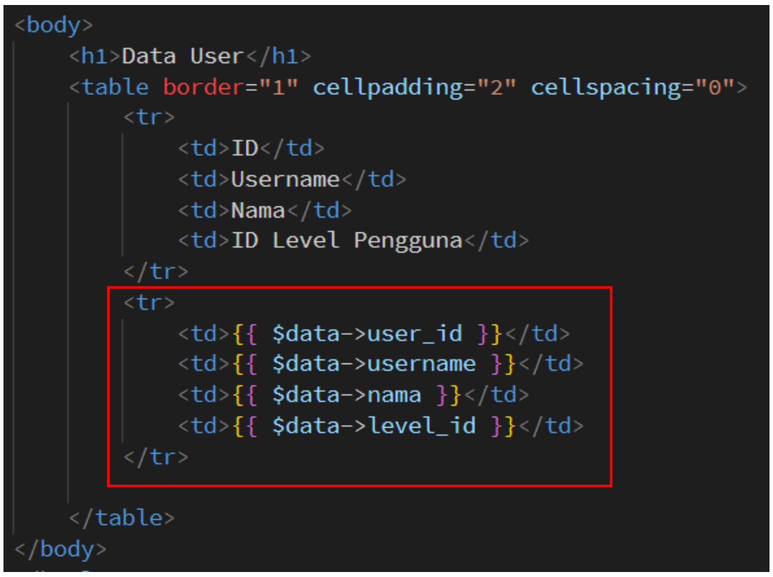
```
$user = UserModel::firstOrCreate(
    [
        'username' => 'manager',
        'nama' => 'Manager',
    ],
);

$user = UserModel::firstOrNew(
    [
        'username' => 'manager',
        'nama' => 'Manager',
    ],
);
```

1. Change the controller file with the name `UserController.php` and change the *script* as shown below



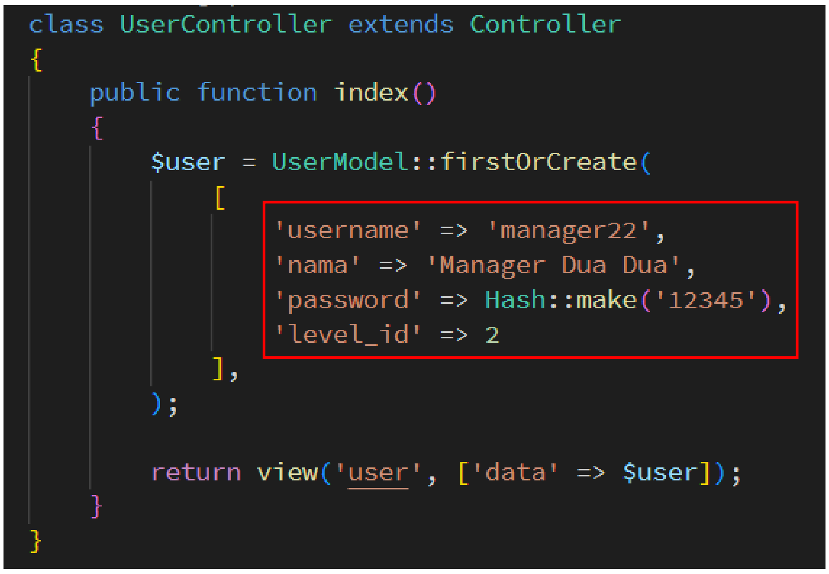
```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstOrCreate(
            [
                'username' => 'manager',
                'nama' => 'Manager',
            ],
        );

        return view('user', ['data' => $user]);
    }
}
```

2. Change the view file `user.blade.php` and change the *script* as shown below



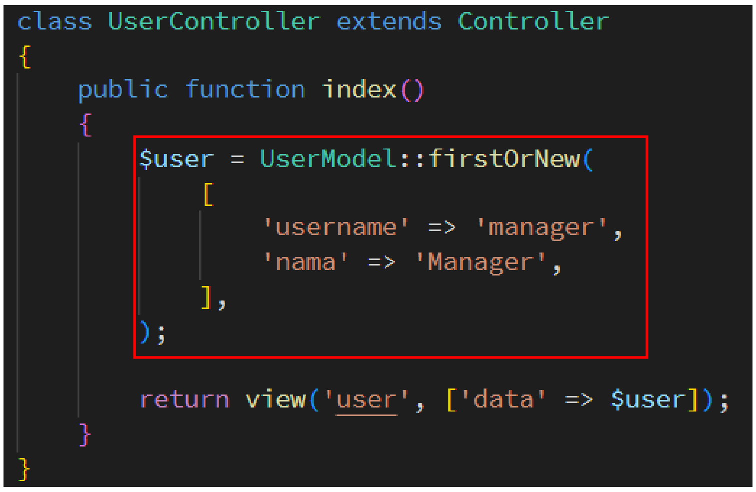
```
<body>
    <h1>Data User</h1>
    <table border="1" cellpadding="2" cellspacing="0">
        <tr>
            <td>ID</td>
            <td>Username</td>
            <td>Nama</td>
            <td>ID Level Pengguna</td>
        </tr>
        <tr>
            <td>{{ $data->user_id }}</td>
            <td>{{ $data->username }}</td>
            <td>{{ $data->nama }}</td>
            <td>{{ $data->level_id }}</td>
        </tr>

    </table>
</body>
```

3. Save the program code of Steps 1 and 2. Then run it in the *browser* and observe what happens and give an explanation in the report
4. Change the controller file with the name `UserController.php` and change the *script* as shown below



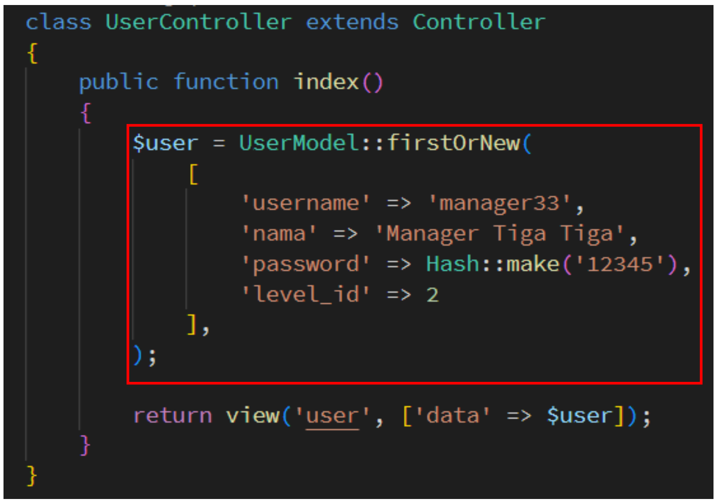
```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstOrCreate(
            [
                'username' => 'manager22',
                'nama' => 'Manager Dua Dua',
                'password' => Hash::make('12345'),
                'level_id' => 2
            ],
        );

        return view('user', ['data' => $user]);
    }
}
```

5. Save the program code Step 4. Then run it in *the browser* and observe what happens and also check *phpMyAdmin* in the m_user table and give an explanation in the report
6. Change the controller file with the name `UserController.php` and change the *script* as shown below



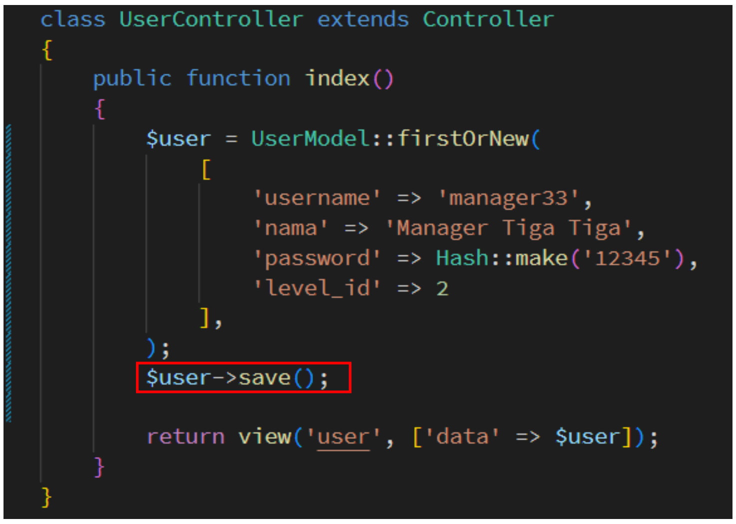
```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstOrNew(
            [
                'username' => 'manager',
                'nama' => 'Manager',
            ],
        );

        return view('user', ['data' => $user]);
    }
}
```

7. Save the program code Step 6. Then run it in the *browser* and observe what happens and give an explanation in the report
8. Change the controller file with the name `UserController.php` and change the *script* as shown below



```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstOrNew(
            [
                'username' => 'manager33',
                'nama' => 'Manager Tiga Tiga',
                'password' => Hash::make('12345'),
                'level_id' => 2
            ],
        );

        return view('user', ['data' => $user]);
    }
}
```

9. Save the program code Step 8. Then run it in *the browser* and observe what happens and also check *phpMyAdmin* in the m_user table and give an explanation in the report
10. Change the controller file with the name `UserController.php` and change the *script* as shown below



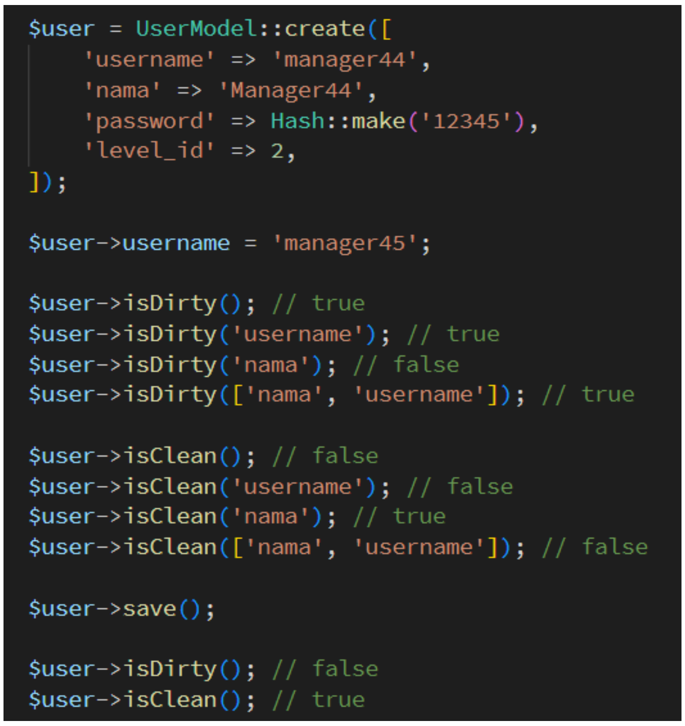
```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::firstOrNew(
            [
                'username' => 'manager33',
                'nama' => 'Manager Tiga Tiga',
                'password' => Hash::make('12345'),
                'level_id' => 2
            ],
        );
        $user->save();

        return view('user', ['data' => $user]);
    }
}
```

11. Save the program code Step 9. Then run it in *the browser* and observe what happens and also check *phpMyAdmin* in the m_user table and give an explanation in the report
12. Report the results of this Practicum-2.4 and *commit* changes to *git*.

Lab Course 2.5 – *Attribute Changes*

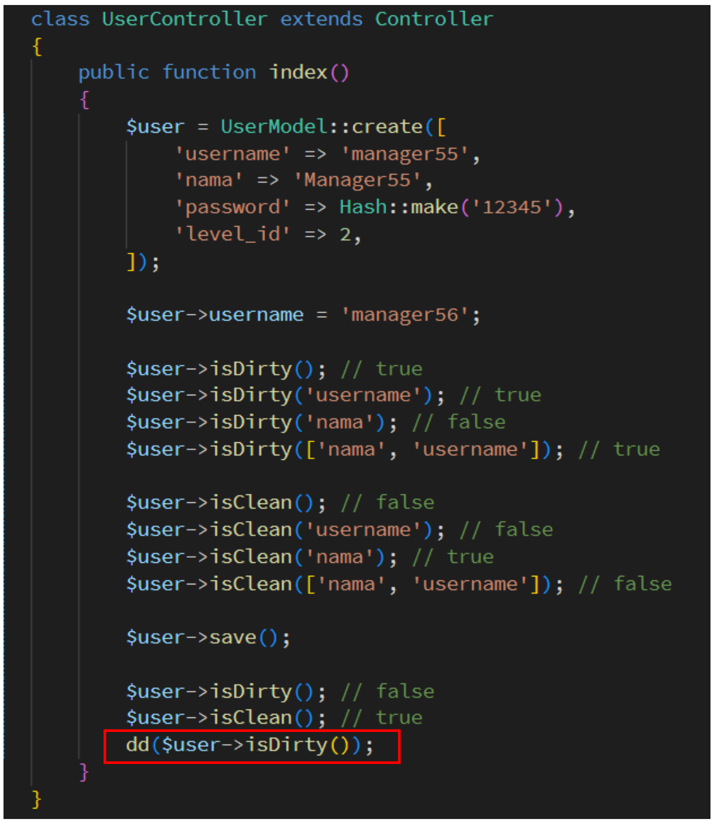
Eloquent provides `isDirty`, `isClean`, and `wasChanged methods` to examine the internal state of your model and determine how its attributes have changed since the model was first retrieved.

The `isDirty` method determines whether any model attributes have changed since the model was retrieved. You can pass a specific attribute name or set of attributes to the `isDirty` method to determine if any attribute is "dirty". The `isClean method` will determine whether an attribute has remained unchanged since the model was retrieved. This method also accepts optional attribute arguments:



```php
$user = UserModel::create([
    'username' => 'manager44',
    'nama' => 'Manager44',
    'password' => Hash::make('12345'),
    'level_id' => 2,
]);

$user->username = 'manager45';

$user->isDirty(); // true
$user->isDirty('username'); // true
$user->isDirty('nama'); // false
$user->isDirty(['nama', 'username']); // true

$user->isClean(); // false
$user->isClean('username'); // false
$user->isClean('nama'); // true
$user->isClean(['nama', 'username']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

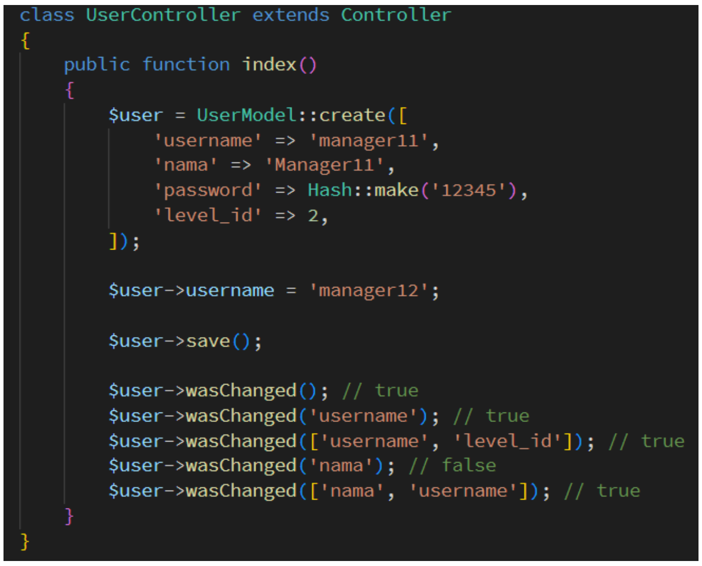
1. Change the controller file with the name `UserController.php` and change the *script* as shown below.



```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::create([
            'username' => 'manager55',
            'nama' => 'Manager55',
            'password' => Hash::make('12345'),
            'level_id' => 2,
        ]);

        $user->username = 'manager56';

        $user->isDirty(); // true
        $user->isDirty('username'); // true
        $user->isDirty('nama'); // false
        $user->isDirty(['nama', 'username']); // true

        $user->isClean(); // false
        $user->isClean('username'); // false
        $user->isClean('nama'); // true
        $user->isClean(['nama', 'username']); // false

        $user->save();

        $user->isDirty(); // false
        $user->isClean(); // true
        dd($user->isDirty());
    }
}
```

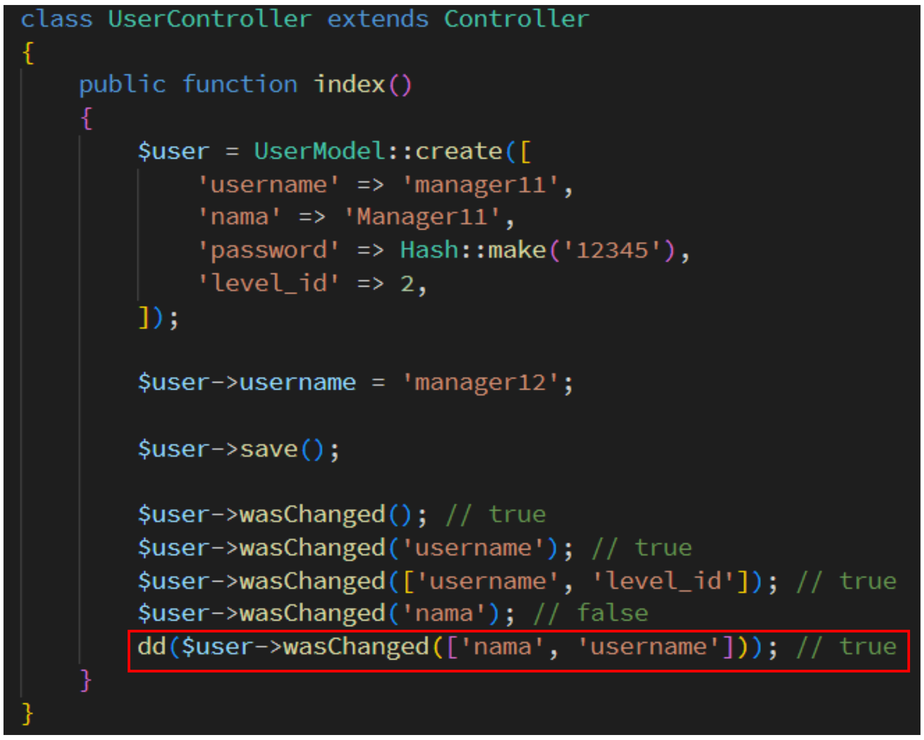
2. Save the program code Step 1. Then run it in the *browser* and observe what happens and give an explanation in the report

   This wasChanged method determines whether any attributes were changed when the model was last stored in the current request cycle. If needed, you can provide an attribute name to see if a specific attribute has changed:



```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::create([
            'username' => 'manager11',
            'nama' => 'Manager11',
            'password' => Hash::make('12345'),
            'level_id' => 2,
        ]);

        $user->username = 'manager12';

        $user->save();

        $user->wasChanged(); // true
        $user->wasChanged('username'); // true
        $user->wasChanged(['username', 'level_id']); // true
        $user->wasChanged('nama'); // false
        $user->wasChanged(['nama', 'username']); // true
    }
}
```

3. Change the controller file with the name `UserController.php` and change the *script* as shown below



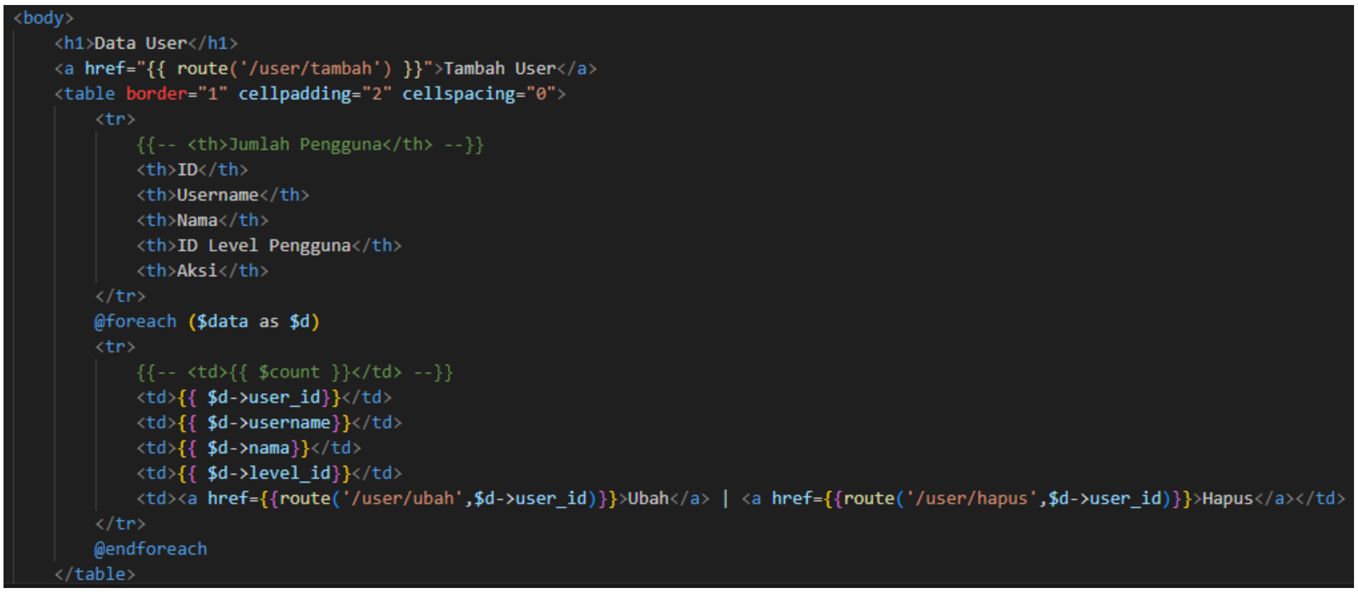
```php
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::create([
            'username' => 'manager11',
            'nama' => 'Manager11',
            'password' => Hash::make('12345'),
            'level_id' => 2,
        ]);

        $user->username = 'manager12';

        $user->save();

        $user->wasChanged(); // true
        $user->wasChanged('username'); // true
        $user->wasChanged(['username', 'level_id']); // true
        $user->wasChanged('nama'); // false
        dd($user->wasChanged(['nama', 'username'])); // true
    }
}
```

4. Save the program code Step 3. Then run it in the *browser* and observe what happens and give an explanation in the report
5. Report this Practicum-2.5 result and *commit* changes to *git*.

Praktikum 2.6 – *Create, Read, Update, Delete (CRUD)*

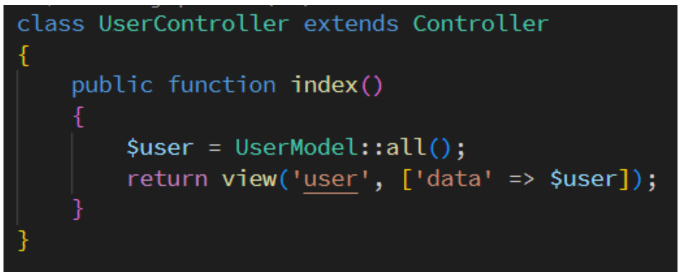
As we already know, CRUD stands for *Create*, *Read*, *Update* and *Delete*. CRUD is a term for the process of processing data in a database, such as inputting data into a database, displaying data from a database, editing data in a database and deleting data from a database. Follow the steps below to perform CRUD with Eloquent.

1. To be able to do CRUD, we will modify the user.blade.php page to have buttons to add, edit and delete data.
   Each button in Laravel will run a specific route. Therefore, before modifying the user.blade.php page we first create a route that will be run when the button is clicked.
   Add *script routes* to the `web.php file` to execute the function when the add button is clicked.
   ```
   Route::get('/user/tambah', [UserController::class, 'tambah'])->name('/user/tambah');
   ```
   Next add *script routes* to the `web.php file` to execute the function when the edit button is clicked.
   ```
   Route::get('/user/ubah/{id}', [UserController::class, 'ubah'])->name('/user/ubah');
   ```
   Finally, add *the routes script* to the `web.php file` to execute the function when the delete button is clicked.
   ```
   Route::get('/user/hapus/{id}', [UserController::class, 'hapus'])->name('/user/hapus');
   ```
   Information:
   In the edit and delete buttons, route is a *route parameter* because edit and delete accept user_id parameters to specify which **user_id** to edit or delete.
2. Open the view file on the `user.blade.php` and create the scritp as below



```
<body>
    <h1>Data User</h1>
    <a href="{{ route('/user/tambah') }}">Tambah User</a>
    <table border="1" cellpadding="2" cellspacing="0">
        <tr>
            {{-- <th>Jumlah Pengguna</th> --}}
            <th>ID</th>
            <th>Username</th>
            <th>Nama</th>
            <th>ID Level Pengguna</th>
            <th>Aksi</th>
        </tr>
        @foreach ($data as $d)
        <tr>
            {{-- <td>{{ $count }}</td> --}}
            <td>{{ $d->user_id}}</td>
            <td>{{ $d->username}}</td>
            <td>{{ $d->nama}}</td>
            <td>{{ $d->level_id}}</td>
            <td><a href={{route('/user/ubah',$d->user_id)}}>Ubah</a> | <a href={{route('/user/hapus',$d->user_id)}}>Hapus</a></td>
        </tr>
        @endforeach
    </table>
```

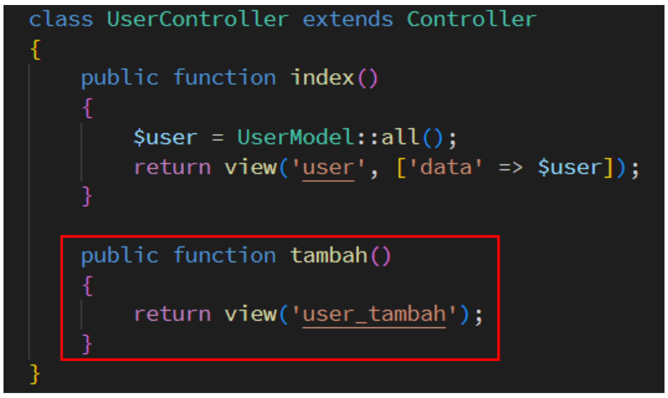
3. Open the controller file on UserController.php and create the script for *read* as below



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::all();
        return view('user', ['data' => $user]);
    }
}
```

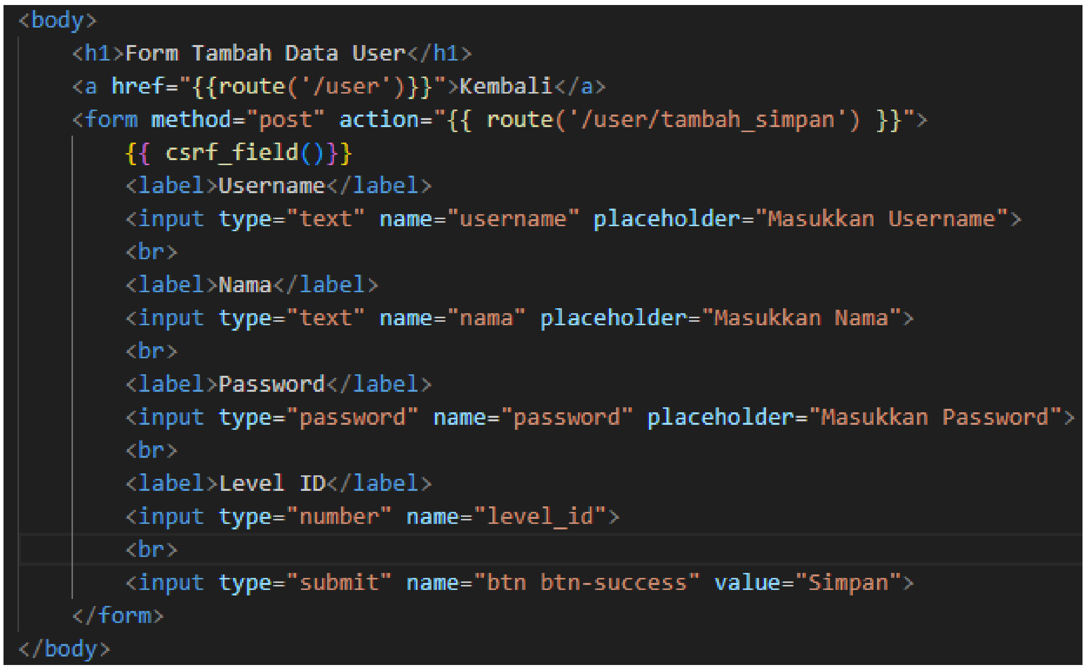
4. Save the program code of Steps 1 and 2. Then run the user page on the /user route in the *browser* and give an explanation in the report.

5. The next step is to create or add user data. When the Add User button is clicked, a page containing the add data form will be displayed. In order to open the add data page, first add the "add" function to the UserController as below.



```
class UserController extends Controller
{
    public function index()
    {
        $user = UserModel::all();
        return view('user', ['data' => $user]);
    }

    public function tambah()
    {
        return view('user_tambah');
    }
}
```

6. Continue by creating a new file in the *view* containing the add data form with the name `user_tambah.blade.php` and create the script as below.

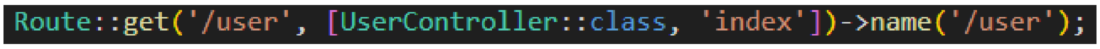


```
<body>
    <h1>Form Tambah Data User</h1>
    <a href="{{route('/user')}}">Kembali</a>
    <form method="post" action="{{ route('/user/tambah_simpan') }}">
        {{ csrf_field()}}
        <label>Username</label>
        <input type="text" name="username" placeholder="Masukkan Username">
        <br>
        <label>Nama</label>
        <input type="text" name="nama" placeholder="Masukkan Nama">
        <br>
        <label>Password</label>
        <input type="password" name="password" placeholder="Masukkan Password">
        <br>
        <label>Level ID</label>
        <input type="number" name="level_id">
        <br>
        <input type="submit" name="btn btn-success" value="Simpan">
    </form>
</body>
```

Information:

- When the Save button is clicked, the form will execute the action. This action will execute the function of storing data in the database and will be called via route.
- There is a Back button that will display the home page again.

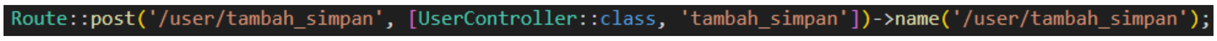
7. Modify the *routes script* in the `web.php` file for the /user route as below.



```
Route::get('/user', [UserController::class, 'index'])->name('/user');
```

8. Save the program code Step 5 to 7. Then run it in *the browser* and click on the link "**+ Add User**" observe what is happening and give an explanation in the report.

9. Next we will add a function to save the data filled in the add data form. First, add a route to the `web.php` file that will perform the function of saving data first.



```
Route::post('/user/tambah_simpan', [UserController::class, 'tambah_simpan'])->name('/user/tambah_simpan');
```

10. Add *a script* to the controller with the file name *UserController.php*. Add a new *function script* with the name tambah_simpan as below screenshot shown.

```
public function tambah_simpan(Request $request){
    UserModel::create([
        'username' => $request->username,
        'nama' => $request->nama,
        'password' => Hash::make($request->password),
        'level_id' => $request->level_id
    ]);
    return redirect('/user');
}
```

11. Save the program code of Steps 9 and 10. Reopen the user list page in the browser, and click the Add User button. Input data on the form and click the save button, then observe what happens and give an explanation in the report

12. The next step is to make *updates* or modify user data. When the edit button is clicked, a new page will open containing the change data form. To open the new page, first add a modify function to the UserController.

```
public function ubah($id)
{
    $user = UserModel::find($id);
    return view('user_ubah', ['data' => $user]);
}
```

13. After that, create a new file in the *view* with the name `user_ubah.blade.php` and create the script as below.
    Information:
    - When the Edit button is clicked, the form will execute the action. This action will execute the function of saving data changes in the database and will be called via route.
    - There is a Back button that will display the home page again.
    - The change data page displays the previous data values stored in the database.

```
<body>
    <h1>Form Ubah Data User</h1>
    <a href="{{route('/user')}}">Kembali</a>
    <br>
    <form method="post" action="{{ route('/user/ubah_simpan',$data->user_id) }}">
        {{ csrf_field() }}
        {{ method_field('PUT') }}

        <label>Username</label>
        <input type="text" name="username" value="{{ $data->username }}">
        <br>

        <label>Nama</label>
        <input type="text" name="nama" value="{{ $data->nama }}">
        <br>

        <label>Level ID</label>
        <input type="number" name="level_id" value="{{ $data->level_id }}">
        <br>

        <input type="submit" name="btn btn-success" value="Ubah">
    </form>
</body>
```

14. Save the program code of Steps 12 and 13. Then run it in the *browser* and click the "**Change**" link, observe what is happening and give an explanation in the report.

15. Next we will add a function to save the data changes to what is filled in the change data form. First, add a route to the `web.php` file that will perform the transform data function first.

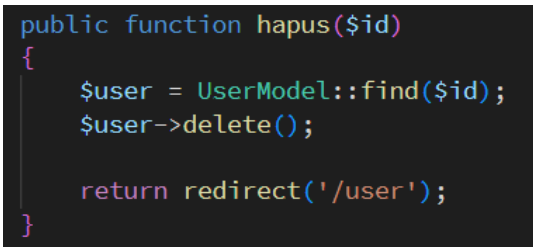
```
Route::put('/user/ubah_simpan/{id}', [UserController::class, 'ubah_simpan'])->name('/user/ubah_simpan');
```

16. Add *a script* to the controller with the file name `UserController.php`. Create a new function named ubah_simpan.

```
public function ubah_simpan($id, Request $request){
    $user = UserModel::find($id);

    $user->username = $request->username;
    $user->nama = $request->nama;
    $user->level_id = $request->level_id;

    $user->save();
    return redirect('/user');
}
```

17. Save the program code of Steps 15 and 16. Reopen the browser's user list page, and click the Change button on one of the lines. Make changes to the data on the form and click the Change button, observe what happens and give an explanation in the report.

18. Now we will add the delete data function. Add *a script* to the controller with the file name `UserController.php`. Add *a script* in the class and create a new method named delete.



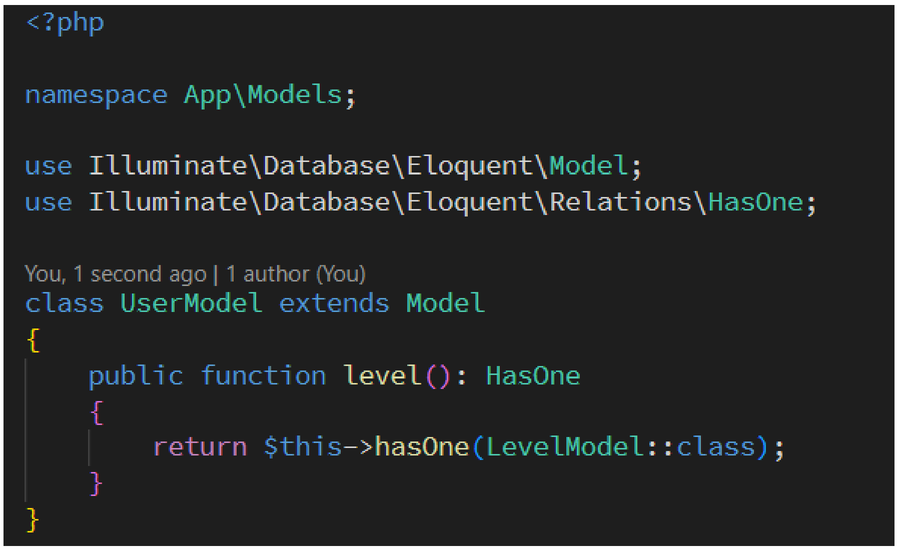
```php
public function hapus($id)
{
    $user = UserModel::find($id);
    $user->delete();

    return redirect('/user');
}
```

19. Save the program code. Reopen the browser's user list page, and click the Delete button on one of the lines. Observe what happened and give explanations in the report.
20. Report this Practicum-2.6 result and *commit* changes to *git*.

Practicum 2.7 – *Relationships*

One to One

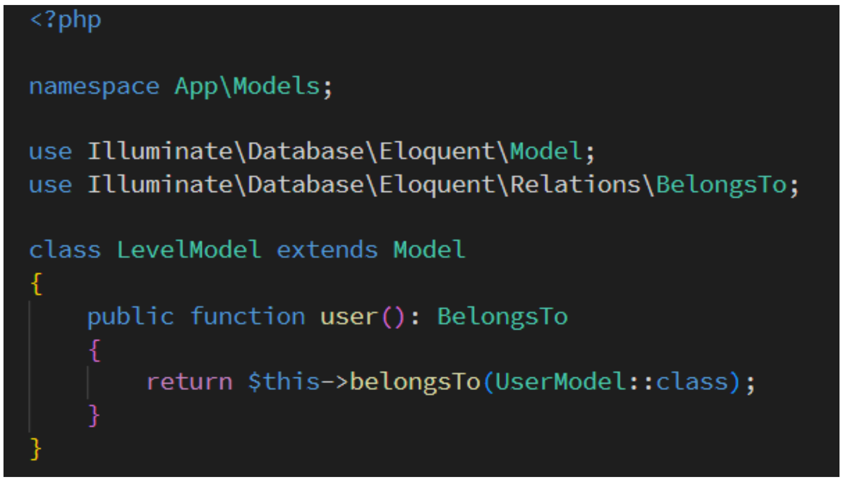
One-to-one relationships are a very basic type of database relationship. For example, a `Usermodel` might be associated with a single Levelmodel model. To define this relationship, we'll place the `Levelmodel` method on the `Usermodel` model. The Levelmodel method should call `the hasOne` method and return the result. This hasOne method is available to your model through the Illuminate`\Database\Eloquent\Model base class`:



```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

You, 1 second ago | 1 author (You)
class UserModel extends Model
{
    public function level(): HasOne
    {
        return $this->hasOne(LevelModel::class);
    }
}
```

Defining the Opposite of a *One-to-one Relationship*

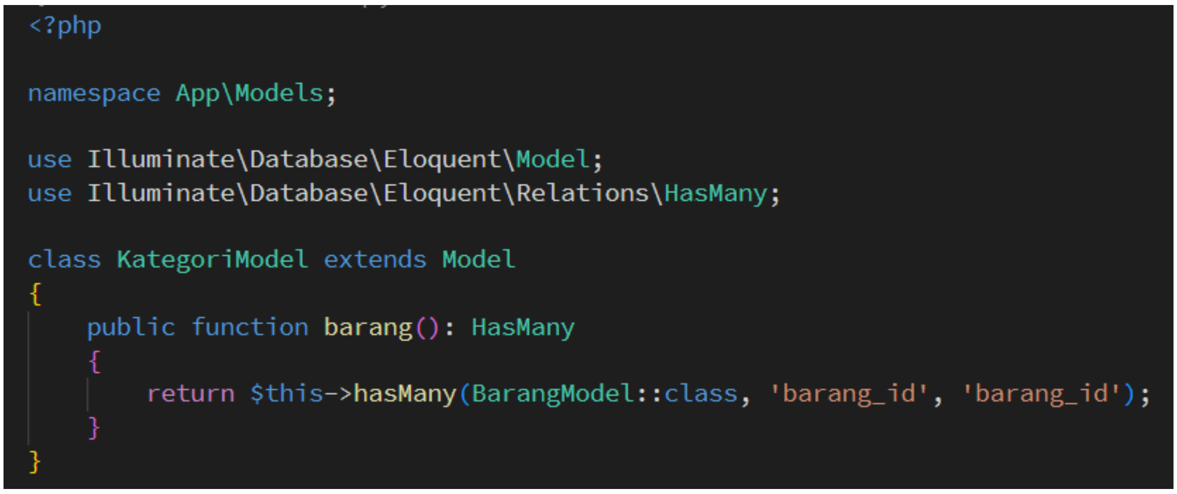
So, we can access the `Levelmodel` model from our `Usermodel` `model`. Next, let's define the relationship on the Levelmodel model that allows us to access the user. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:



```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class LevelModel extends Model
{
    public function user(): BelongsTo
    {
        return $this->belongsTo(UserModel::class);
    }
}
```

One to Many

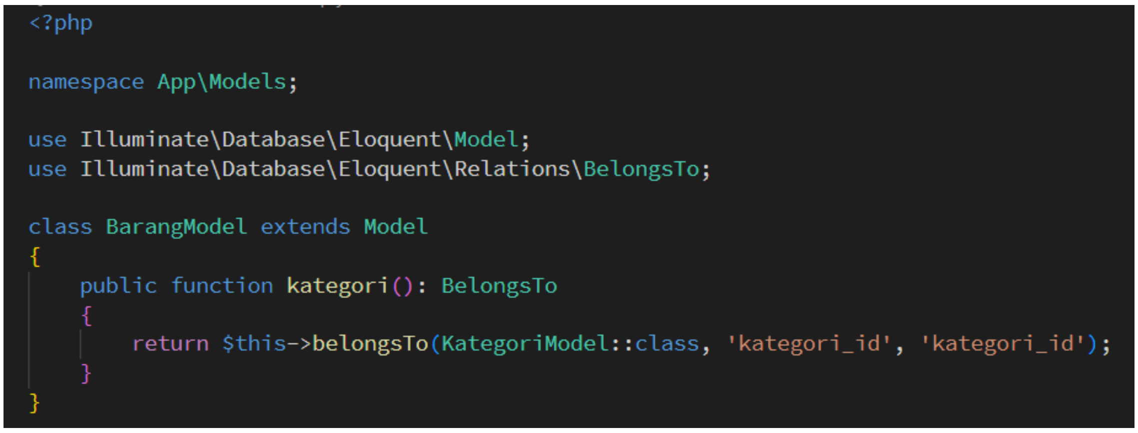
One-to-many relationships are used to define relationships where one model is the parent of one or more derived models. For example, 1 category may have an unlimited number of items. Like all other Eloquent relationships, one-to-many relationships are defined by defining methods on your Eloquent model:



```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class KategoriModel extends Model
{
    public function barang(): HasMany
    {
        return $this->hasMany(BarangModel::class, 'barang_id', 'barang_id');
    }
}
```

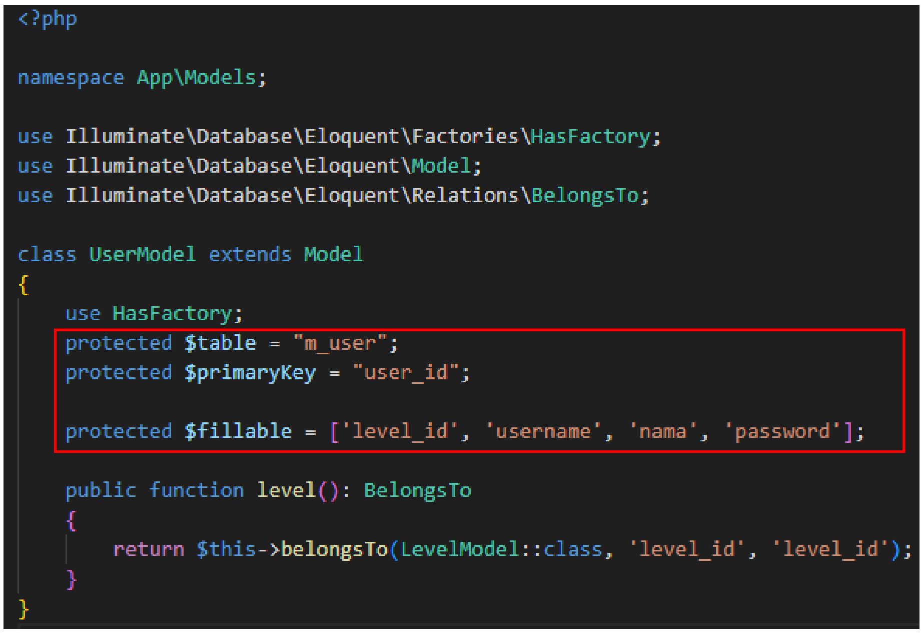
One to Many (Inverse) / Belongs To

Now that we can access all the items, let's define relationships so that the item can access its parent category. To determine the inverse of a `hasMany` relationship, specify the method of the relationship in the child model that calls `the belongsTo`:



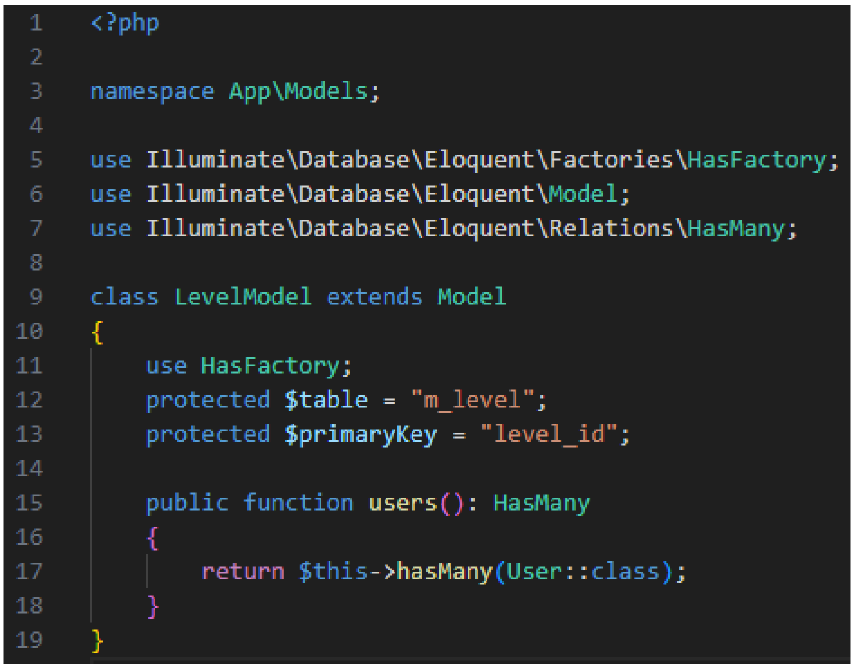
```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class BarangModel extends Model
{
    public function kategori(): BelongsTo
    {
        return $this->belongsTo(KategoriModel::class, 'kategori_id', 'kategori_id');
    }
}
```

1. Open the model file on `UserModel.php` and add the scritp to it as below



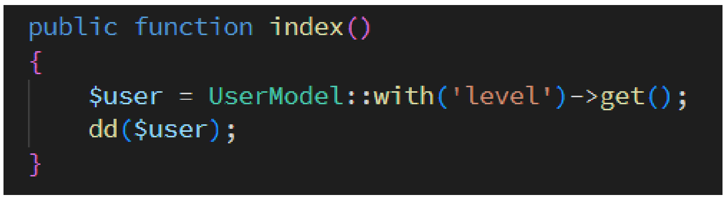
```php
<?php

namespace App\Models;

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

class UserModel extends Model
{
    use HasFactory;
    protected $table = "m_user";
    protected $primaryKey = "user_id";

    protected $fillable = ['level_id', 'username', 'nama', 'password'];

    public function level(): BelongsTo
    {
        return $this->belongsTo(LevelModel::class, 'level_id', 'level_id');
    }
}
```

And in LevelModel.php add the scritp to be as below



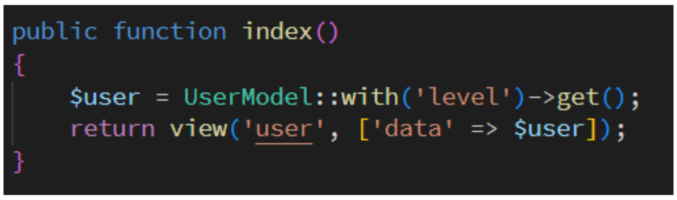
```php
<?php

namespace App\Models;

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

class LevelModel extends Model
{
    use HasFactory;
    protected $table = "m_level";
    protected $primaryKey = "level_id";

    public function users(): HasMany
    {
        return $this->hasMany(User::class);
    }
}
```

2. Open the controller file on UserController.php and change the *script* method to as below



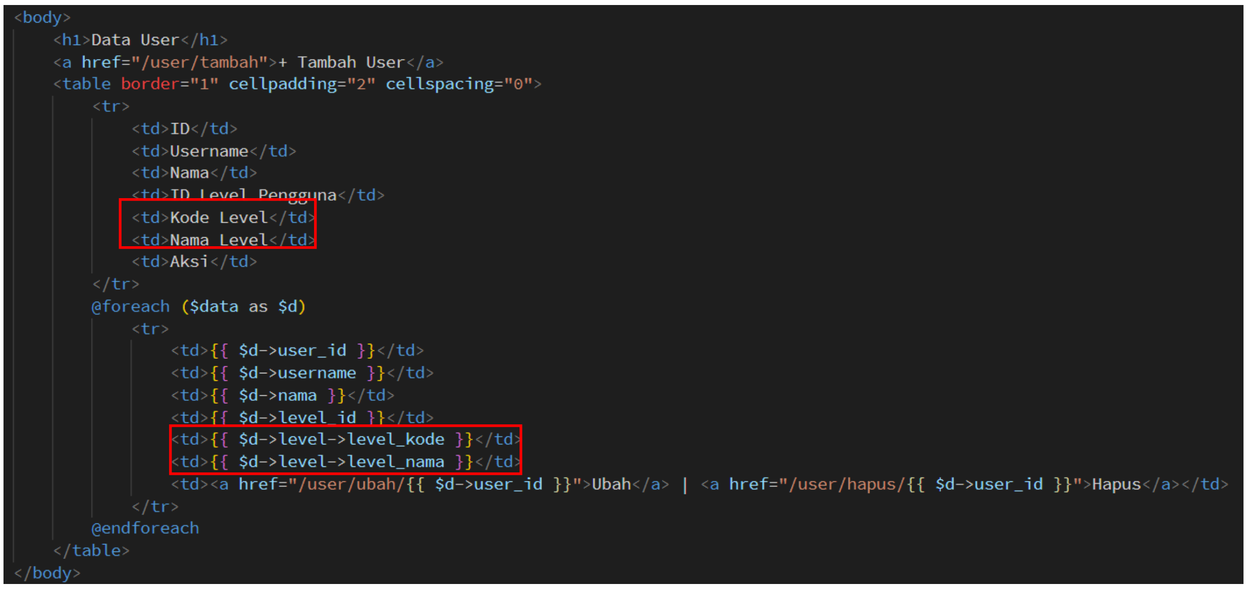
```php
public function index()
{
    $user = UserModel::with('level')->get();
    dd($user);
}
```

3. Save the program code Step 2. Then run the link in the *browser*, then observe what happens and explain it in the report
4. Open the controller file on UserController.php and change the *script* method to as below



```php
public function index()
{
    $user = UserModel::with('level')->get();
    return view('user', ['data' => $user]);
}
```

5. Open the view file on the `user.blade.php` and change the *script* to as below



```
<body>
    <h1>Data User</h1>
    <a href="/user/tambah">+ Tambah User</a>
    <table border="1" cellpadding="2" cellspacing="0">
        <tr>
            <td>ID</td>
            <td>Username</td>
            <td>Nama</td>
            <td>ID Level Pengguna</td>
            <td>Kode Level</td>
            <td>Nama Level</td>
            <td>Aksi</td>
        </tr>
        @foreach ($data as $d)
            <tr>
                <td>{{ $d->user_id }}</td>
                <td>{{ $d->username }}</td>
                <td>{{ $d->nama }}</td>
                <td>{{ $d->level_id }}</td>
                <td>{{ $d->level->level_kode }}</td>
                <td>{{ $d->level->level_nama }}</td>
                <td><a href="/user/ubah/{{ $d->user_id }}">Ubah</a> | <a href="/user/hapus/{{ $d->user_id }}">Hapus</a></td>
            </tr>
        @endforeach
    </table>
</body>
```

6. Save the program code of Steps 4 and 5. Then run the link in the *browser*, then observe what happens and explain it in the report
7. Report the results of this Practicum-2.7 and *commit* changes to *git*.

*Thank you, and good luck ****