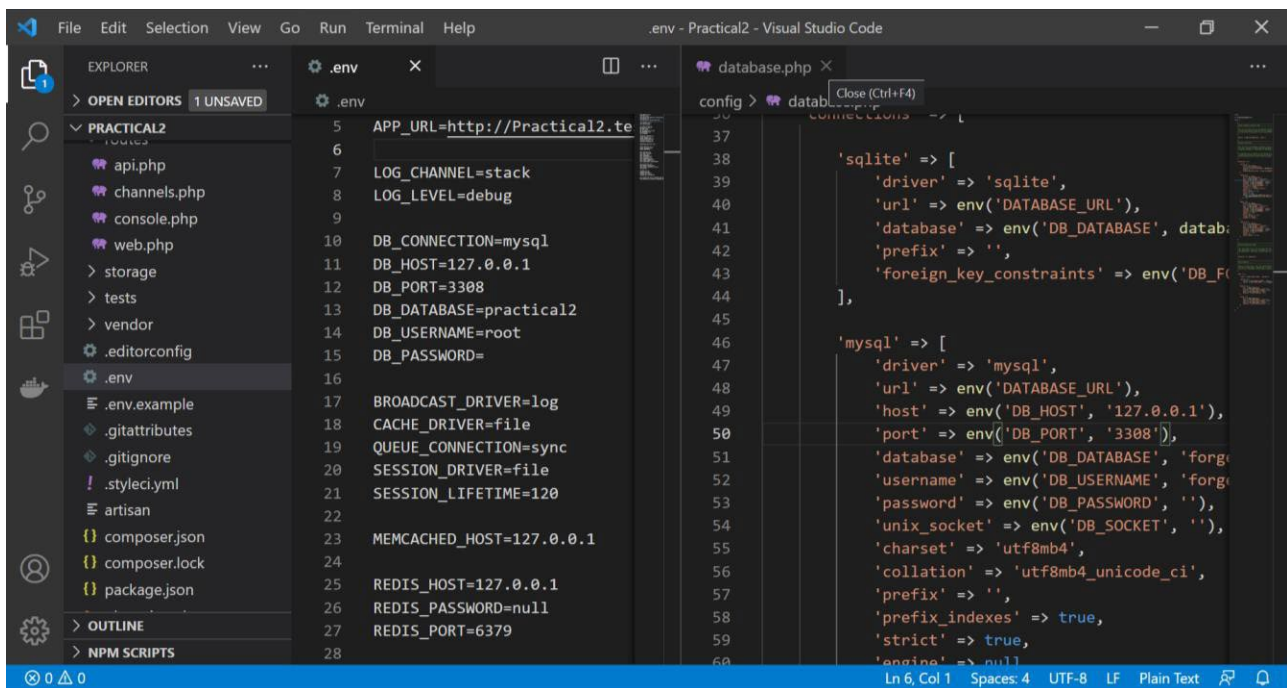


Practical 3: Working with Databases and Object-Relational Mapping (Part I)

In this lab, you will use previously developed web application to learn expound concept of database and object-relational mapping (ORM).

1 Database Configuration and Fetch Data using Controller

Database configuration file reflects the configurations in `.env`. General specification such as database name, host, username and password can be further specified through `.env` file in the Laravel web application to match with the server that it intends to connect as shown in Figure 1 and Figure 2.



The screenshot shows the Visual Studio Code editor with two files open: `.env` and `database.php`. The `.env` file contains the following configuration:

```
5 APP_URL=http://Practical2.test
6
7 LOG_CHANNEL=stack
8 LOG_LEVEL=debug
9
10 DB_CONNECTION=mysql
11 DB_HOST=127.0.0.1
12 DB_PORT=3308
13 DB_DATABASE=practical2
14 DB_USERNAME=root
15 DB_PASSWORD=
16
17 BROADCAST_DRIVER=log
18 CACHE_DRIVER=file
19 QUEUE_CONNECTION=sync
20 SESSION_DRIVER=file
21 SESSION_LIFETIME=120
22
23 MEMCACHED_HOST=127.0.0.1
24
25 REDIS_HOST=127.0.0.1
26 REDIS_PASSWORD=null
27 REDIS_PORT=6379
28
```

The `database.php` file shows the database configuration for Laravel:

```
37
38
39 'sqlite' => [
40     'driver' => 'sqlite',
41     'url' => env('DATABASE_URL'),
42     'database' => env('DB_DATABASE', database_path('database.sqlite')),
43     'prefix' => '',
44     'foreign_key_constraints' => env('DB_FOREIGN_KEYS'),
45 ],
46
47 'mysql' => [
48     'driver' => 'mysql',
49     'url' => env('DATABASE_URL'),
50     'host' => env('DB_HOST', '127.0.0.1'),
51     'port' => env('DB_PORT', '3308'),
52     'database' => env('DB_DATABASE', 'forge'),
53     'username' => env('DB_USERNAME', 'forge'),
54     'password' => env('DB_PASSWORD', ''),
55     'unix_socket' => env('DB_SOCKET', ''),
56     'charset' => 'utf8mb4',
57     'collation' => 'utf8mb4_unicode_ci',
58     'prefix' => '',
59     'prefix_indexes' => true,
60     'strict' => true,
61     'engine' => null,
62 ]
```

Figure 1: Database configuration file and lists for Laravel Web Application's database.

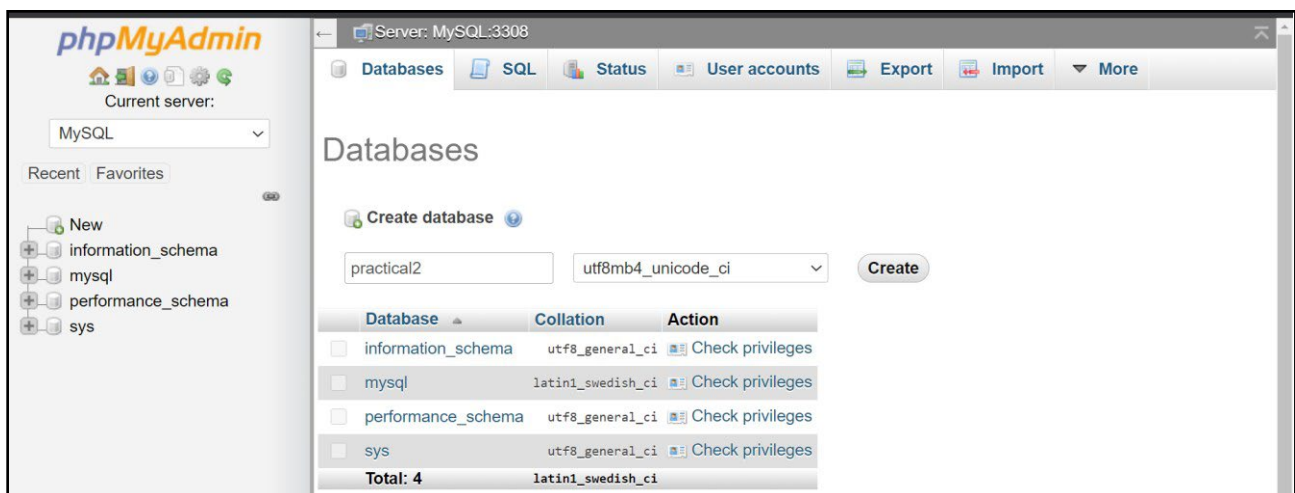
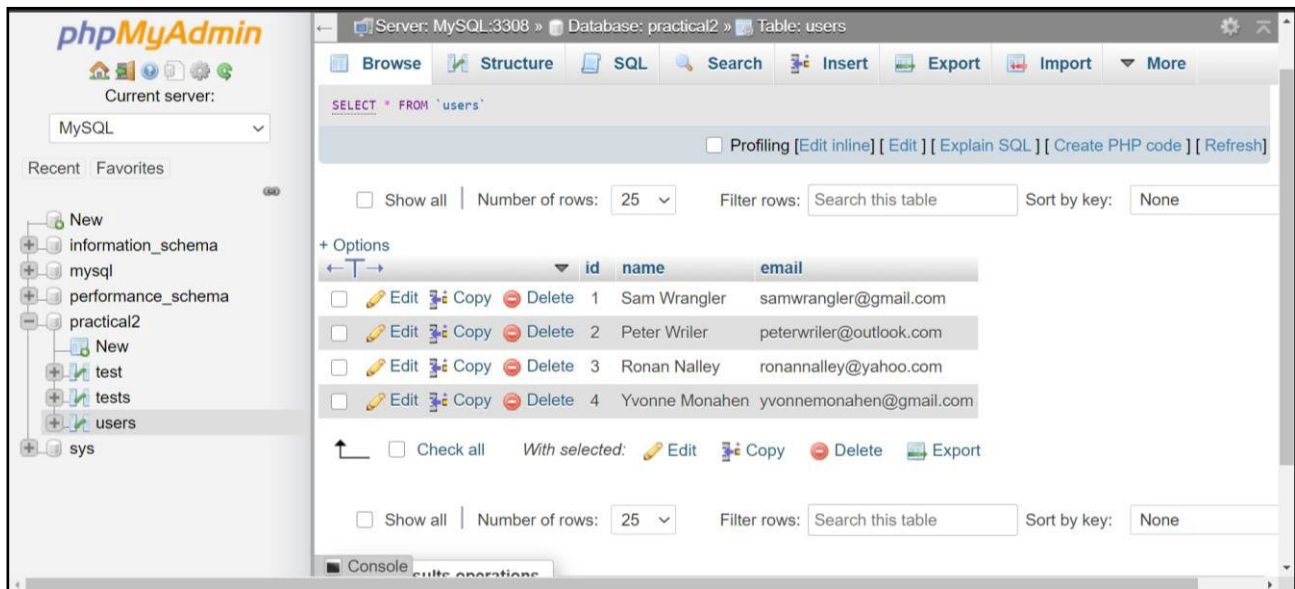


Figure 2: MySQL database to be used by Laravel Web Application.

In order to expound interacts with the web application's database, create a new database as illustrated in above Figure 2. Then, create a users table and insert some data into the table for testing, as shown in Figure 3.



Server: MySQL:3308 » Database: practical2 » Table: users

SELECT * FROM `users`

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key: None

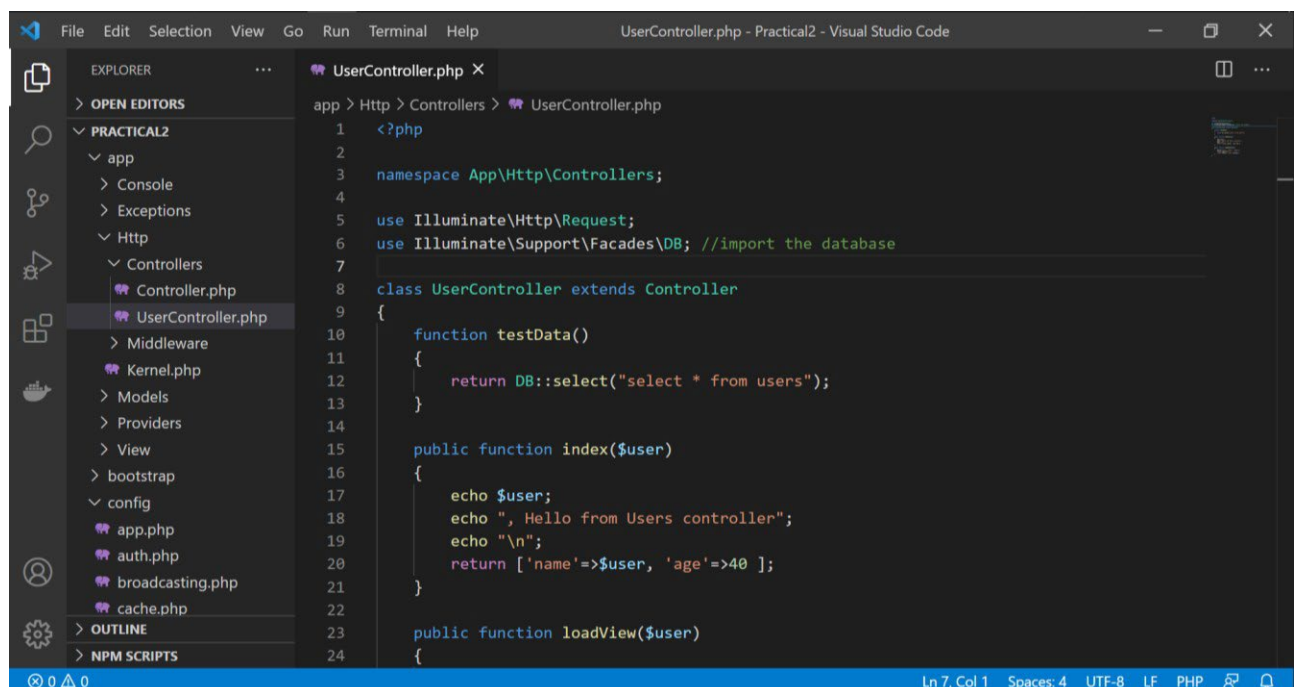
	id	name	email
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	1	Sam Wrangler	samwrangler@gmail.com
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	2	Peter Wriker	peterwriker@outlook.com
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	3	Ronan Nalley	ronannalley@yahoo.com
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	4	Yvonne Monahen	yvonnemonahen@gmail.com

Check all | With selected: ☐ Edit ☐ Copy ☐ Delete ☐ Export

Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key: None

Figure 3: A users database table.

Next, we explore concept of raw SQL queries in Laravel Framework using Controller. Create a simple data fetch function and output with raw SQL queries within the **UserController** created previously as shown in Figure 4.



```

1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use Illuminate\Support\Facades\DB; //import the database
7
8  class UserController extends Controller
9  {
10     function testData()
11     {
12         return DB::select("select * from users");
13     }
14
15     public function index($user)
16     {
17         echo $user;
18         echo ", Hello from Users controller";
19         echo "\n";
20         return ['name'=>$user, 'age'=>40 ];
21     }
22
23     public function loadView($user)
24     {

```

Figure 4: Raw SQL queries for data fetch using a controller.

After having the simple data fetch function and output in **UserController**, create a route to it so that the output can be seen in view result of the web application as shown in Figure 5.

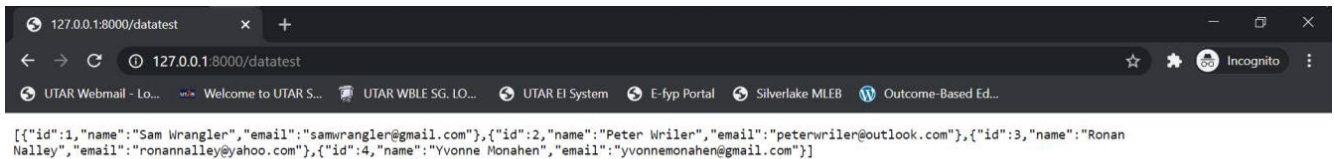


Figure 5: Fetch data from practical 2 database users table using controller.

2 Model

Model is the interface for database of any MVC architecture. Model basically fetch requirement from Controller and model the required data by fetching data from database. Laravel Model contains connection of Laravel web application with database, eloquent object-relational mapping (ORM), database structure and application logics.

Eloquent ORM feature in Laravel Framework enables Laravel web application to map database table with class name. A general rule of class naming for eloquent ORM to be done is: plural name for database table, while singular name for model class name. For instance, database name “**users**” will imply that the model name is “**user**” and if database name “**employees**” will imply that the model name is “**employee**”. In case if a web developer insist on same name for database table and model class, a further configuration is required.

In order to explore the model component in Laravel application, let’s create a model in Laravel web application to interact with **users** database table.

How to create a model in Laravel web application? There are two ways for doing so.

- (a) Through the Artisan CLI “**php artisan make:model**”. Figure 6 shows the example of creating “**User**” model.

```
Microsoft Windows [Version 10.0.19042.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\looyi>cd Desktop
C:\Users\looyi\Desktop>cd test
C:\Users\looyi\Desktop\test>cd Practical2
C:\Users\looyi\Desktop\test\Practical2>php artisan make:model User
Model already exists!
C:\Users\looyi\Desktop\test\Practical2>php artisan make:model User
Model created successfully.
C:\Users\looyi\Desktop\test\Practical2>
```

Figure 6: Using Artisan CLI to create User model.

- (b) Through manual “New File” creation within web application project folder.

Once the model is created, the model file can be found within the Models folder in **app/Models** directory.

Take note: older Laravel versions placed Models folder in app/http/models directory.

Within the newly created User model, create a simple data fetching function from **UserController** as illustrated in Figure 7.

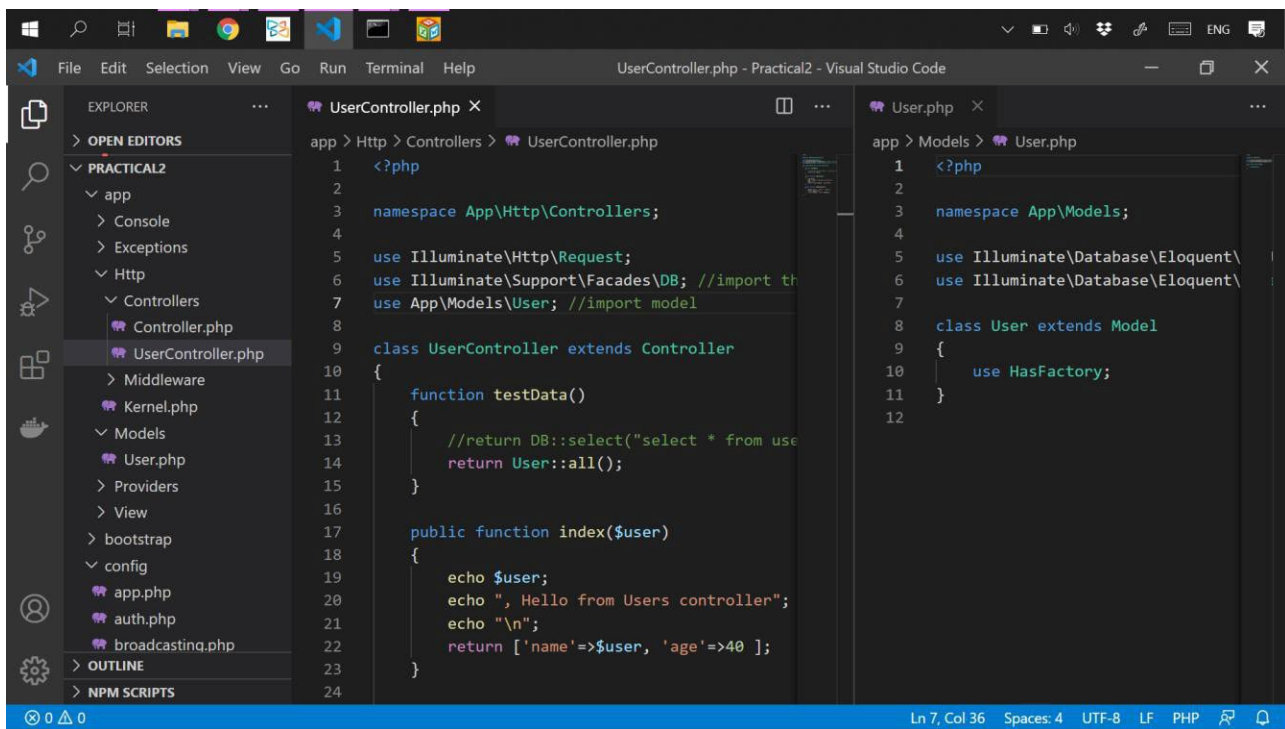


Figure 7: An echoing function in Users Controller.

Route to the controller and see that the data is automatically fetched from Users database table from User Model.

If in case a web developer insist on ignoring the eloquent ORM rule of Laravel framework (having the same name for database table and model class), connection to the database table from model class can still be done through a further configuration by adding **Public \$table="user";** in the model class as shown in Figure 8.

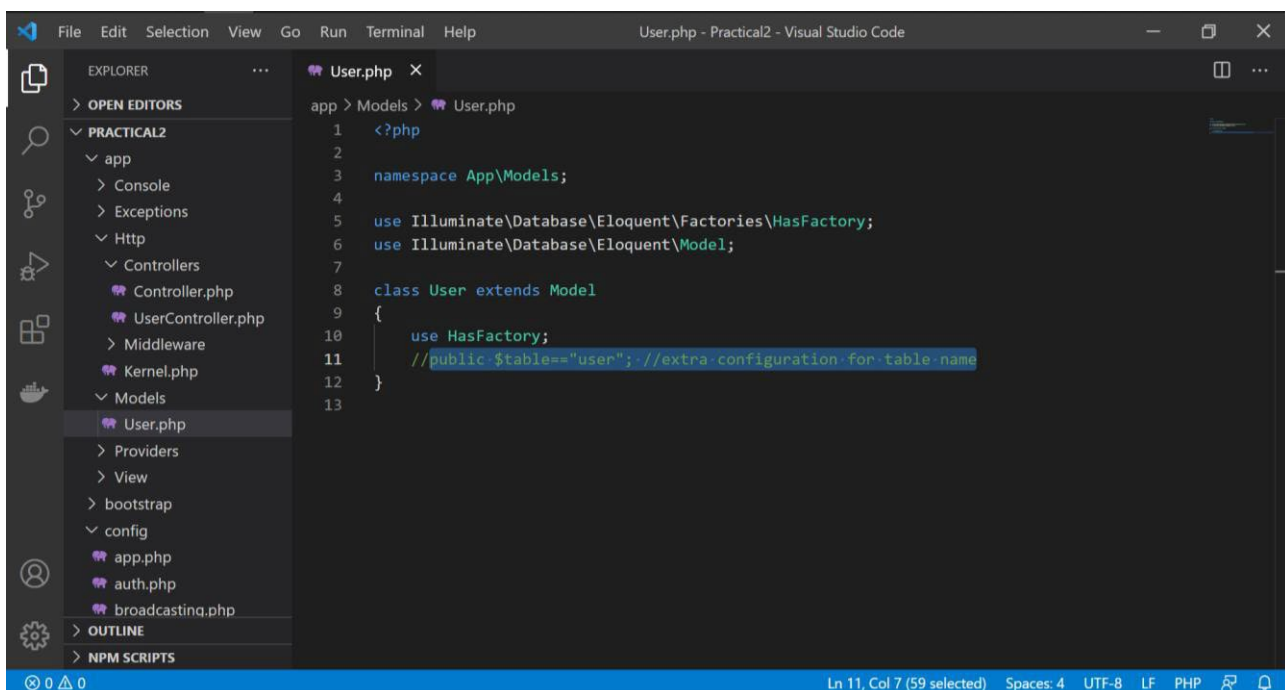


Figure 8: Extra configuration in model class for manual table name mapping.

Thus far, we've explored on fetching data from database using model as interface between controller and database. The output of the data fetch was shown using a return function in controller. In the following session of the practical, we will look into outputting the fetched data to view.

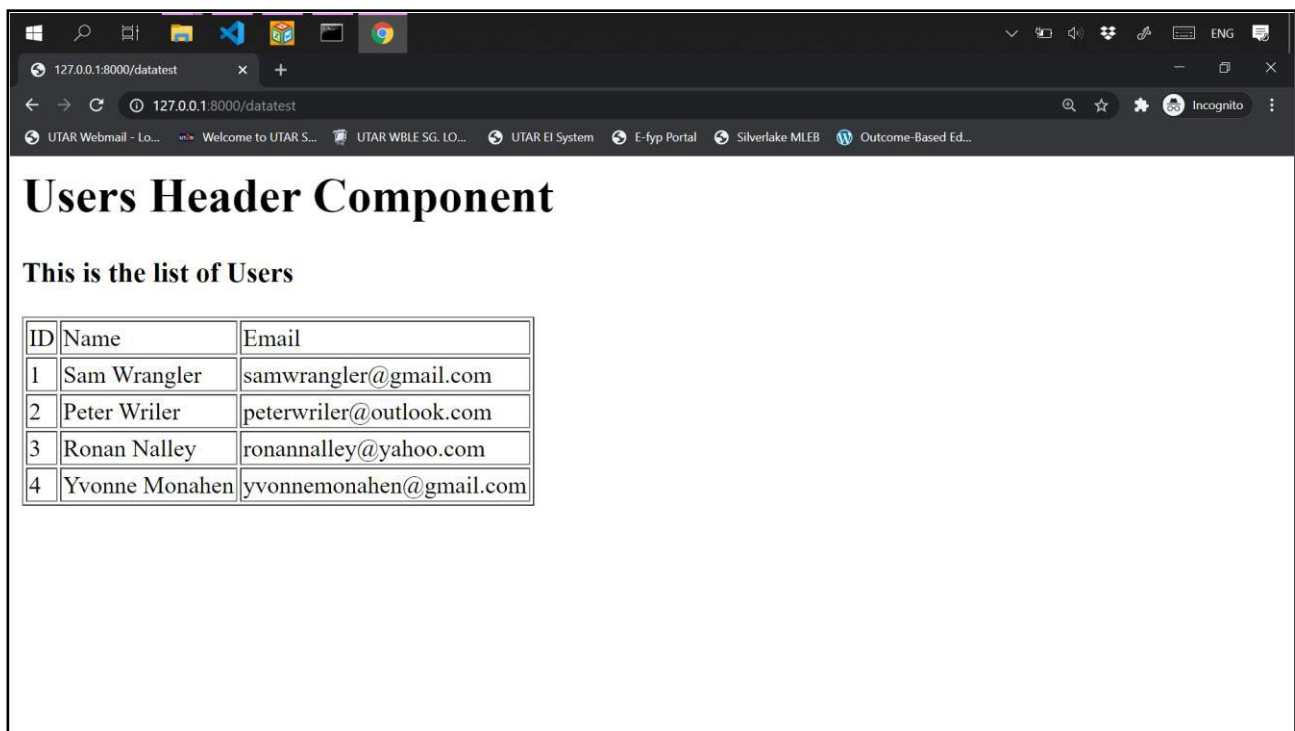
2.1 Showing a list of users to view

Within the controller, we know that “**User::all()**” contains all data from **Users** table. Previously, we used controller to return a view and passed some data to the view for output.

Exercise: Based on the understanding and knowledge thus far;

1. Modify ‘**userInner**’ view to show a table; to list out the ‘**id**’, ‘**name**’ and ‘**email address**’ data.
2. Modify ‘**user**’ view in order to temporarily skip executing other php commands except for including ‘**userInner**’ view.
3. In controller, modify the **testData()** function to pass data into the returning view.

Your output can be as similar / as shown in figure below.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/datatest'. The page content is titled 'Users Header Component' and includes the text 'This is the list of Users'. Below this text is a table with the following data:

ID	Name	Email
1	Sam Wrangler	samwrangler@gmail.com
2	Peter Wrieler	peterwrieler@outlook.com
3	Ronan Nalley	ronannalley@yahoo.com
4	Yvonne Monahen	yvonnemonahen@gmail.com

2.2 Use of Pagination

In scenarios where the list of data is a lot, Laravel Framework offers a feature called 'pagination' to preset the number of data displayed in one page.

In order to see the impact of having pagination, let's add data to the Users table so that it contains at least 15 data.

Use **paginate()** in User table initialization within **UserController** instead of **all()** to enable pagination as shown in Figure 9. Then, referring to the same Figure 9, modify the view to enable navigation to display the rest of the users that are in second and third page.

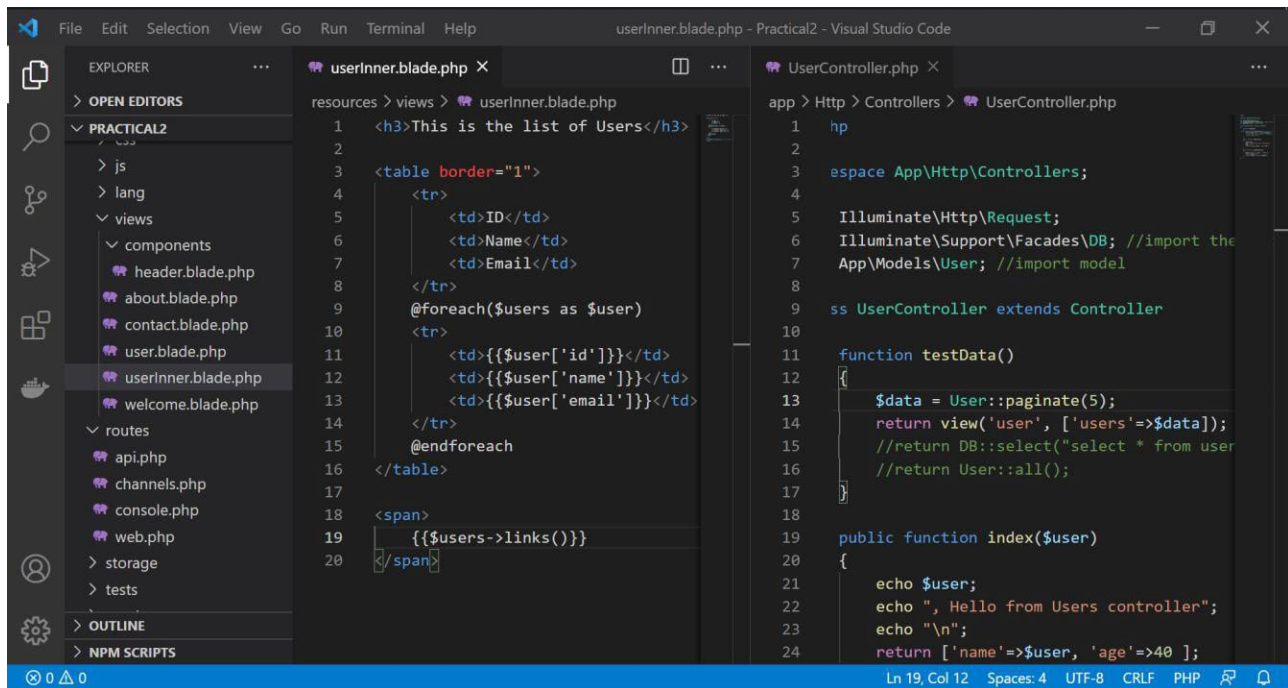


Figure 9: Applying pagination and displaying outputs in different pages.

As the result of applying pagination, the view should be as shown in Figure 10.

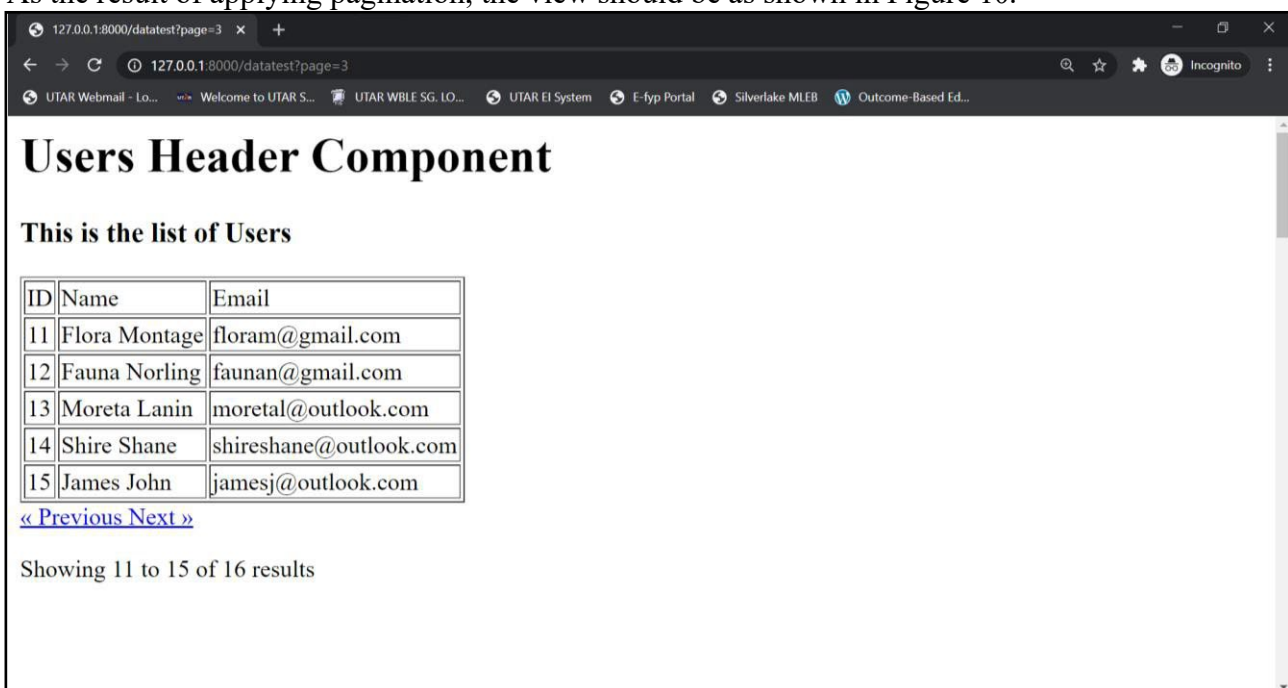


Figure 10: List of users with pagination.

There is a mishandling of CSS within Laravel framework. In order to fix the mishandling, include following scripts to the view file.

```
<style>
    .w-5{
        display: none
    }
</style>
```

Thus far, we've explored how data can be modelled using model then displayed to a view that is invoked by a controller. Additional use of pagination to format the output data into different pages was explored as well. In the following session, let's explore on adding data into database.

2.3 Creating/Inserting Data into Database

Firstly, let's create an interface for inputs; create a new **addUser** view with a simple form for name and email inputs which look like Figure 11. Then, create a view route for **addUser**.

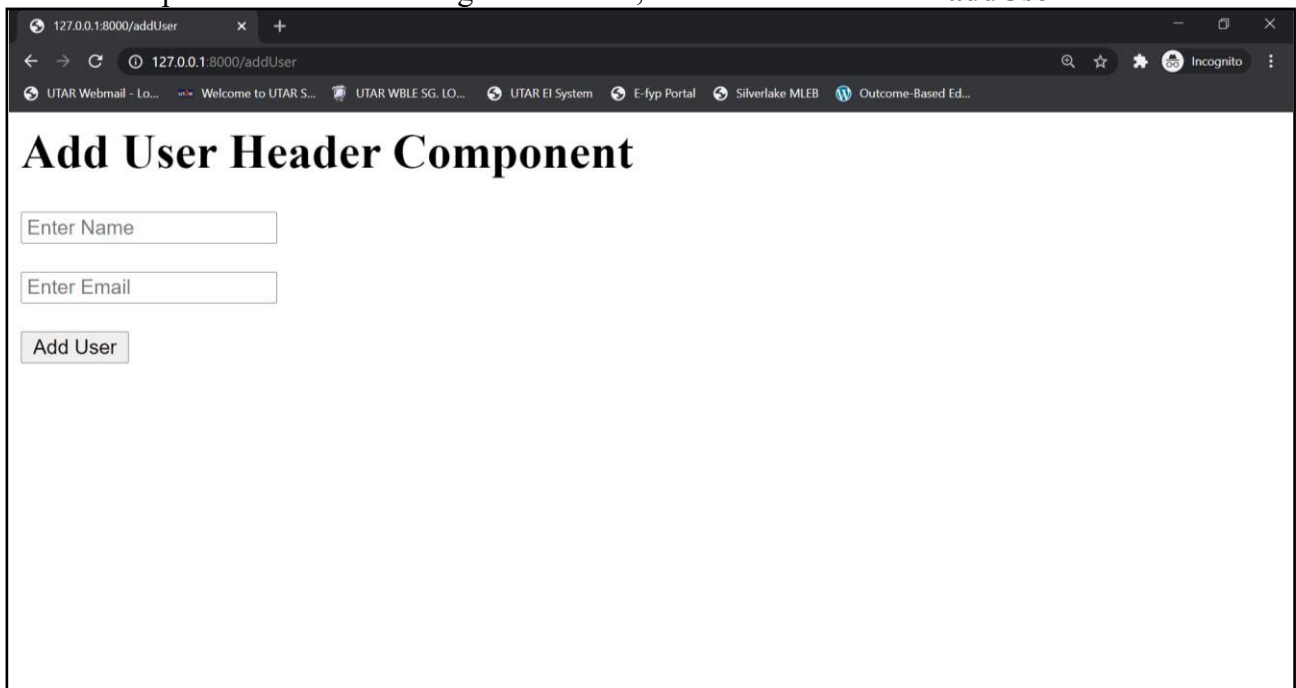
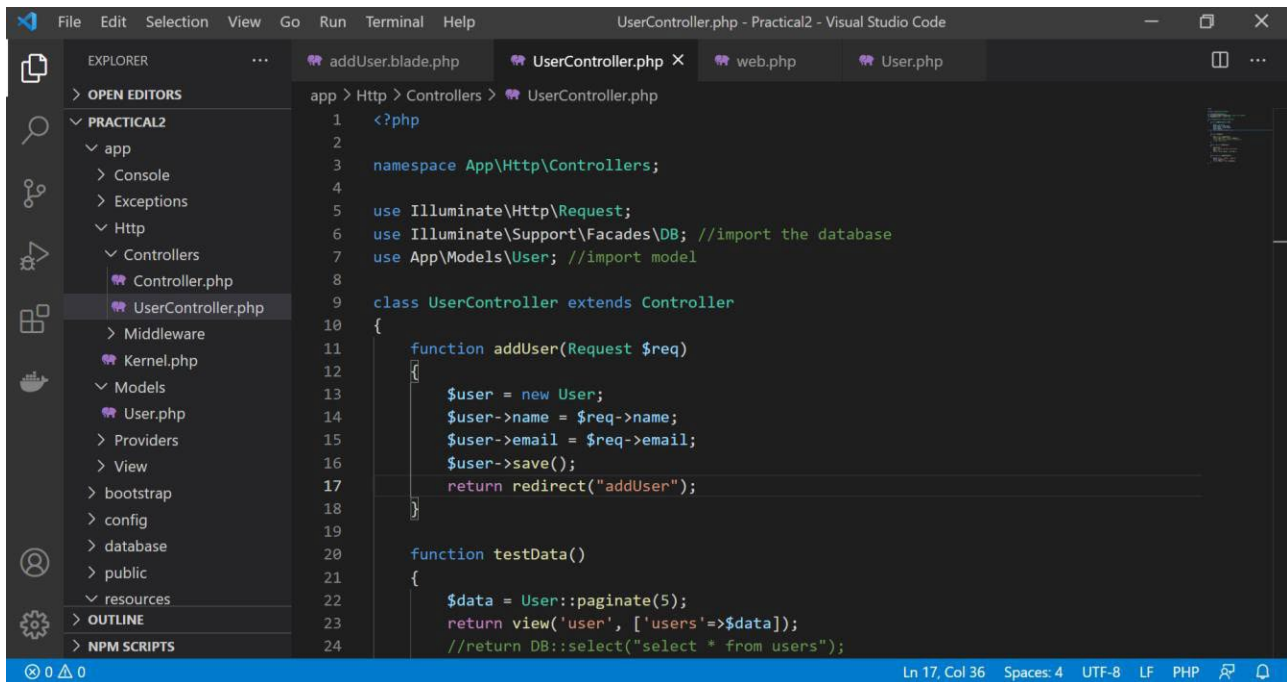
A screenshot of a web browser window. The address bar shows '127.0.0.1:8000/addUser'. The browser has several tabs open, including 'UTAR Webmail - Lo...', 'Welcome to UTAR S...', 'UTAR WBLE SG. LO...', 'UTAR EI System', 'E-lyp Portal', 'Silverlake MLEB', and 'Outcome-Based Ed...'. The main content area of the browser displays a form titled 'Add User Header Component'. The form consists of two text input fields, one labeled 'Enter Name' and the other 'Enter Email', stacked vertically. Below these fields is a button labeled 'Add User'.

Figure 11: A simple form View for inputs.

Secondly, create an **addUser** function in **UserController** to fetch data input from the form as shown in Figure 12. Then, create a controller route to the controller's function.

The image is a screenshot of the Visual Studio Code editor. The Explorer sidebar on the left shows a project structure with folders like 'app', 'Http', 'Controllers', 'Middleware', 'Kernel.php', 'Models', 'Providers', 'View', 'bootstrap', 'config', 'database', 'public', 'resources', 'OUTLINE', and 'NPM SCRIPTS'. The 'Controllers' folder is expanded, showing 'UserController.php'. The main editor window displays the code for 'UserController.php'. The code starts with a PHP tag, namespace declaration 'App\Http\Controllers', and imports for 'Illuminate\Http\Request', 'Illuminate\Support\Facades\DB', and 'App\Models\User'. A class 'UserController' extends 'Controller'. It contains two methods: 'addUser' which takes a 'Request \$req' and creates a new 'User' object with 'name' and 'email' from the request, saves it, and redirects to 'addUser'; and 'testData' which paginates users and returns a view. The status bar at the bottom indicates 'Ln 17, Col 36', 'Spaces: 4', 'UTF-8', 'LF', 'PHP', and a search icon.

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6 use Illuminate\Support\Facades\DB; //import the database
7 use App\Models\User; //import model
8
9 class UserController extends Controller
10 {
11     function addUser(Request $req)
12     {
13         $user = new User;
14         $user->name = $req->name;
15         $user->email = $req->email;
16         $user->save();
17         return redirect("addUser");
18     }
19
20     function testData()
21     {
22         $data = User::paginate(5);
23         return view('user', ['users'=>$data]);
24         //return DB::select("select * from users");
25     }
26 }
```

Figure 12: addUser function in UserController to process inputs.

Thirdly, ensure that User class has a public declaration of false for timestamps “**public \$timestamps = false**” as Laravel expects every input to be accompanied by data of “**updated_at**” and “**created_at**”, in which we can declare as false for Users table does not have this two fields.

Apart from outputting data to view, creation / insertion of data into database, deletion of data is a crucial function to be made available in a web application. Data deletion will be explored in the following session of practical.

2.4 Deleting Data from Database

Looking back at the previous view for paginating all data from Users database table, add another column to the table to contain a “Delete” anchor as shown in Figure 13. Clicking the anchor will enable user to delete data of the same row.

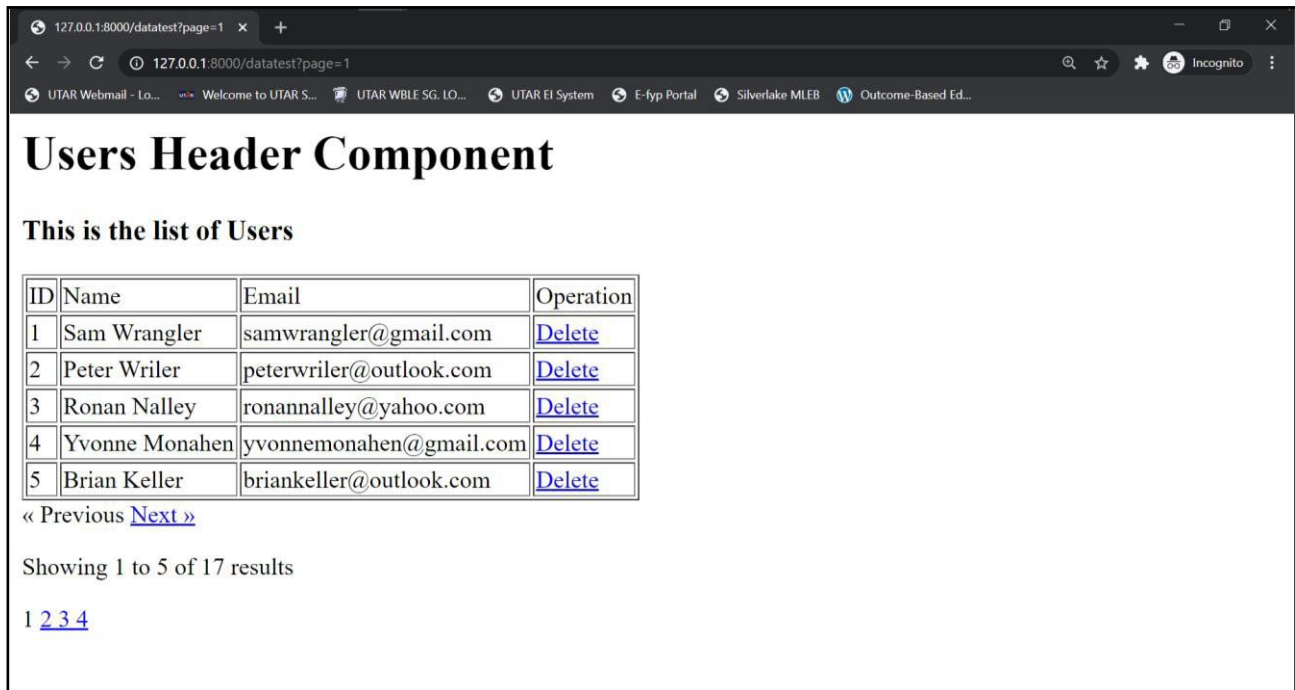


Figure 13: Delete function in userInner view for data deletion.

Then, create a **deleteUser** function in controller as well as the route to the controller as shown in Figure 14.

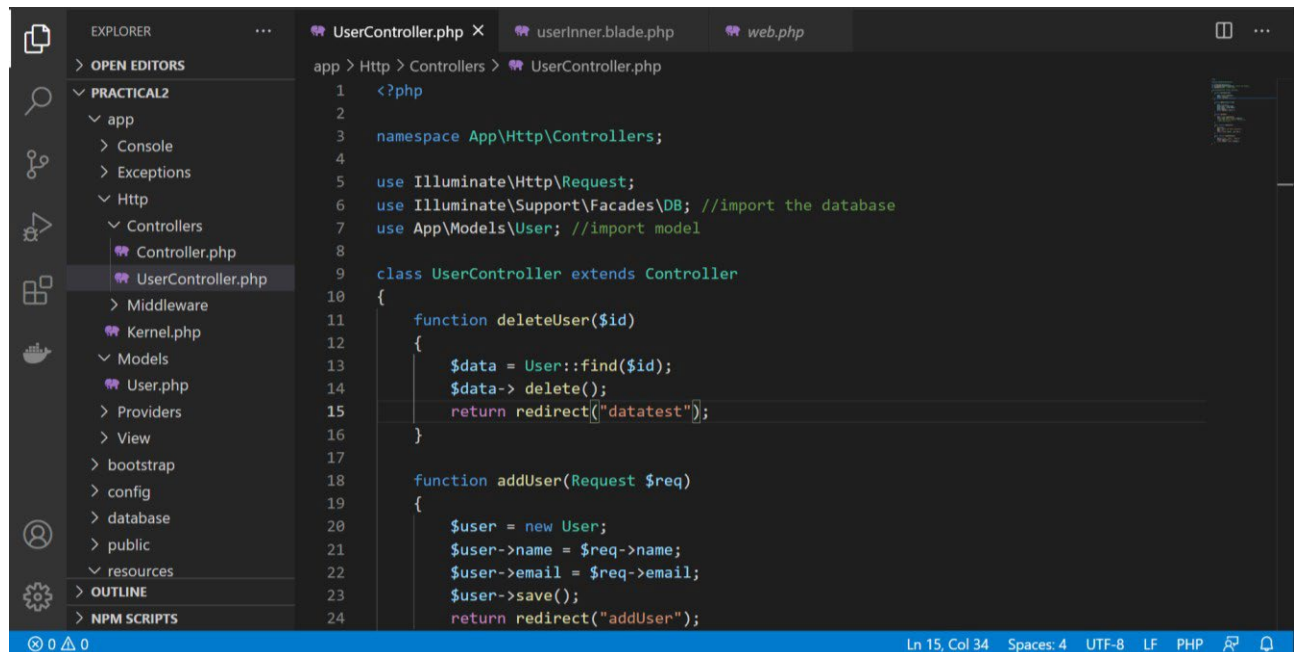


Figure 14: deleteUser function in UserController to process data deletion.

The final function that need to be included in a web application that deals with database will be updating existing data in a database. The following session will explore update of a data in Users database table.

2.5 Updating / Editing Data in Database

Looking back at the previous view for deleting data from **Users** database table, add another column to the table to contain an “**Update**” anchor as shown in Figure 15. Clicking the anchor will enable invoke a form for user to update / edit the existing data.

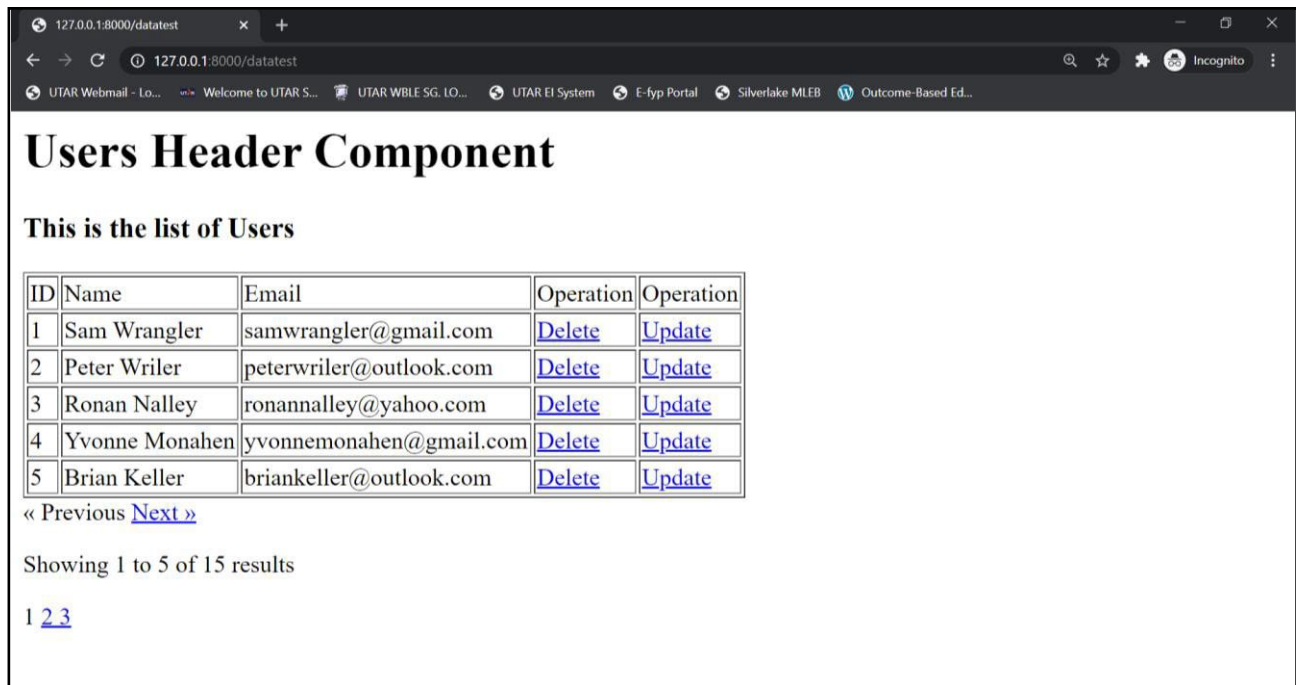


Figure 14: updateUser function in userInner view for data update.

Then create an “**updateUser**” form view, which will be invoked from the event of “**Update**” anchor being clicked to fetch data to be used to update a specific existing data as shown in Figure 15.

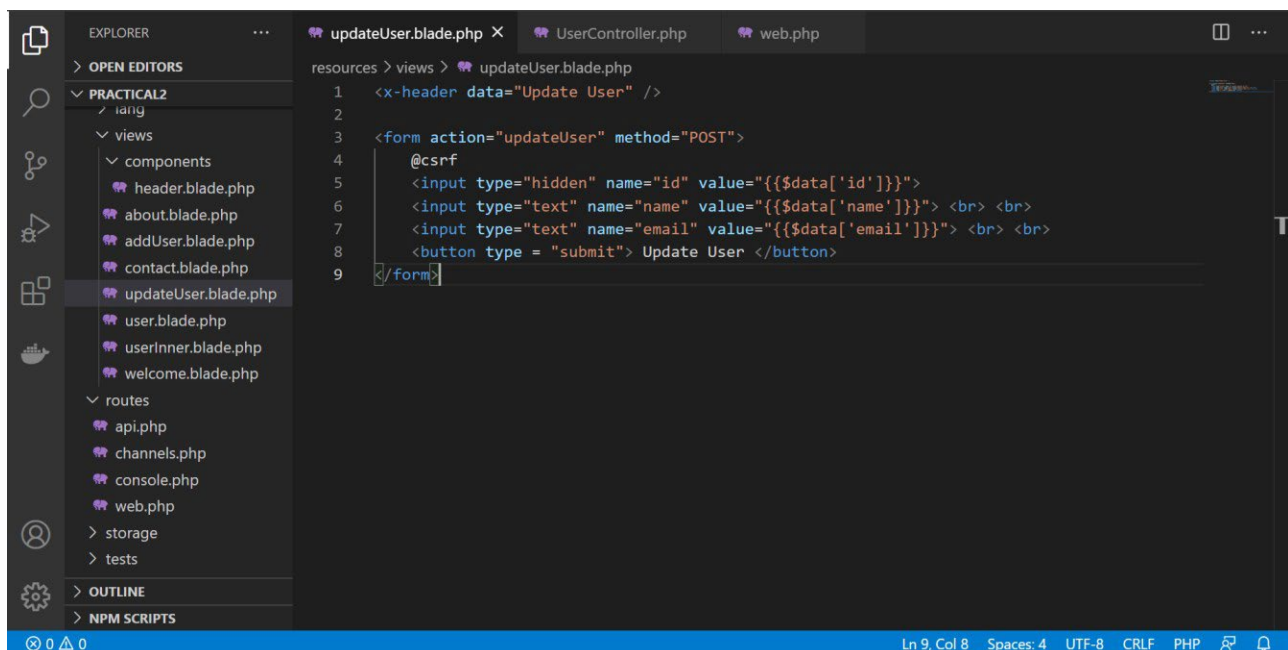


Figure 15: updateUser form view for data update.

Then, create two controller functions; one for showing the data selected for update into **updateUser** form, another one for updating logics as shown in Figure 16.

```
8
9 class UserController extends Controller
10 {
11     function updateUser(Request $req)
12     {
13         $data = User::find($req->id);
14         $data->name = $req->name;
15         $data->email = $req->email;
16         $data->save();
17         return redirect("datatest");
18     }
19
20     function showUpdate($id)
21     {
22         $data = User::find($id);
23         return view("updateUser", ['data'=>$data]);
24     }
25
26     function deleteUser($id)
27     {
28         $data = User::find($id);
29         $data-> delete();
30         return redirect("datatest");
31     }
32 }
```

Figure 16: updateUser logics in UserController.

Finally, create the routes to the **UserController** show form and update logic functions as shown in Figure 17.

```
9     return view('welcome',['username'=>$username]);
10 });
11 */
12 //Route::get("users", "Users@index"); // Older Laravel versions
13 Route::get("users/{user}",[UserController::class,'index']); // Laravel 8
14 Route::get("users/{user}",[UserController::class,'loadView']); // Laravel 8
15 Route::get("datatest",[UserController::class,'testData']);
16 Route::view("addUser", "addUser");
17 Route::post("addUser",[UserController::class,'addUser']);
18 Route::get("deleteUser/{id}",[UserController::class,'deleteUser']);
19 Route::get("updateUser/{id}",[UserController::class,'showUpdate']);
20 Route::post("updateUser/{id}",[UserController::class,'updateUser']);
21
22 Route::get('/', function () {
23     //return view('welcome');
24     return redirect("about");
25 });
26
27 Route::get('/about', function () {
28     return view('about');
29 });
30
31 Route::view("contact","contact");
32 //Short syntax
```

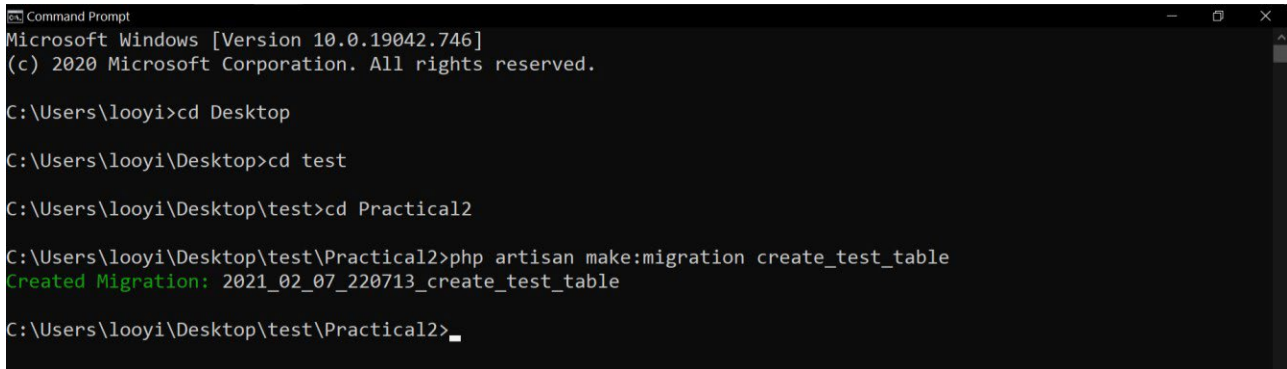
Figure 17: updateUser routes to UserController.

Thus far, main CRUD was explored on an existing database table “Users”. The following practical session expound on the concept of data migration.

3 Data Migration in Laravel

Migration is a feature provided by Laravel framework for automating creation of database table. In order to explore the concept, create a new database table through Artisan CLI as shown in Figure 18.

php artisan make:migration create_test_table



```
Microsoft Windows [Version 10.0.19042.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\looyi>cd Desktop

C:\Users\looyi\Desktop>cd test

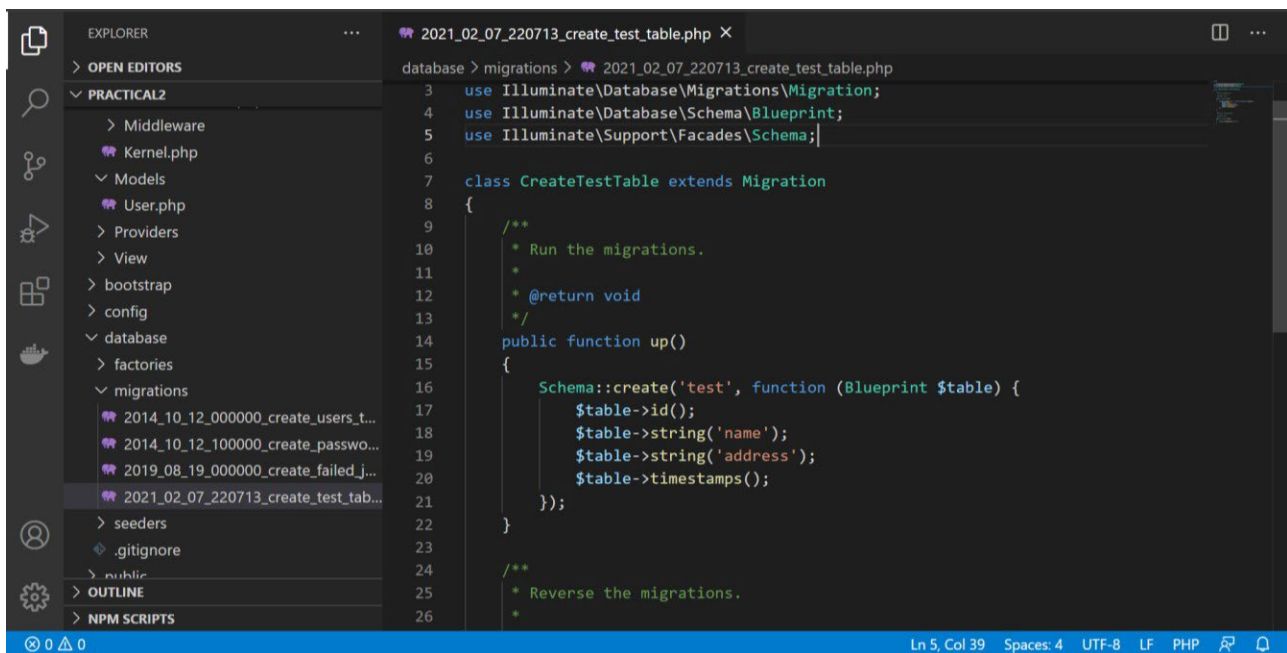
C:\Users\looyi\Desktop\test>cd Practical2

C:\Users\looyi\Desktop\test\Practical2>php artisan make:migration create_test_table
Created Migration: 2021_02_07_220713_create_test_table

C:\Users\looyi\Desktop\test\Practical2>_
```

Figure 18: Data migration initiation using Artisan CLI.

The migration file created is located in **\Database\Migrations**. Within the migration file, further define the database table structure as shown in Figure 19.



```
database > migrations > 2021_02_07_220713_create_test_table.php
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  class CreateTestTable extends Migration
8  {
9      /**
10       * Run the migrations.
11       *
12       * @return void
13       */
14      public function up()
15      {
16          Schema::create('test', function (Blueprint $table) {
17              $table->id();
18              $table->string('name');
19              $table->string('address');
20              $table->timestamps();
21          });
22      }
23
24      /**
25       * Reverse the migrations.
26       */
27  }
```

Figure 19: test database table structure.

In order to automate the creation of the table, execute Artisan CLI migrate command as shown in Figure 20.

php artisan migrate --path=/database/migrations/migration_name.php



```
C:\Users\looyi\Desktop\test\Practical2>php artisan migrate --path=/database/migrations/2021_02_07_220713_create_test_table.php
Migrating: 2021_02_07_220713_create_test_table
Migrated: 2021_02_07_220713_create_test_table (31.96ms)

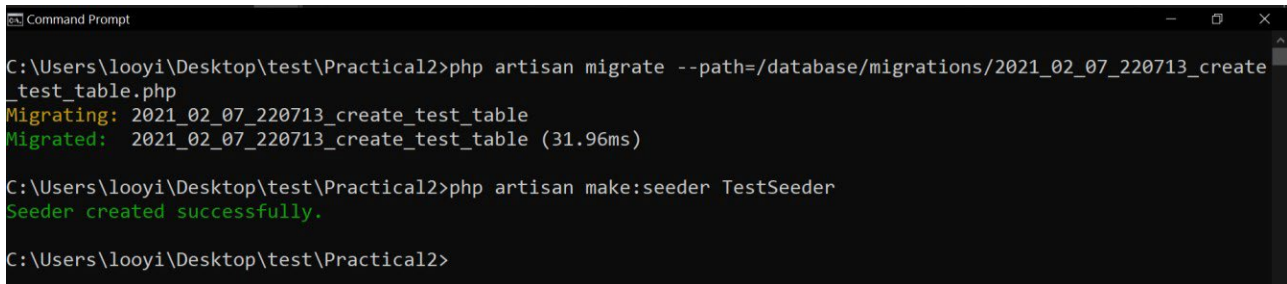
C:\Users\looyi\Desktop\test\Practical2>_
```

Figure 20: Migrate specific table with Artisan CLI.

Migration only create database table structure but not data. Following practical session expound on the automation of data creation with Laravel data seeding feature.

4 Data Seeding in Laravel

The concept of data seeding is adding dummy data into a database table, in which, is a good practice for testing purpose. Data seeding file need to be first created using Artisan CLI as shown in Figure 21.



```
C:\Users\looyi\Desktop\test\Practical2>php artisan migrate --path=/database/migrations/2021_02_07_220713_create_test_table.php
Migrating: 2021_02_07_220713_create_test_table
Migrated: 2021_02_07_220713_create_test_table (31.96ms)

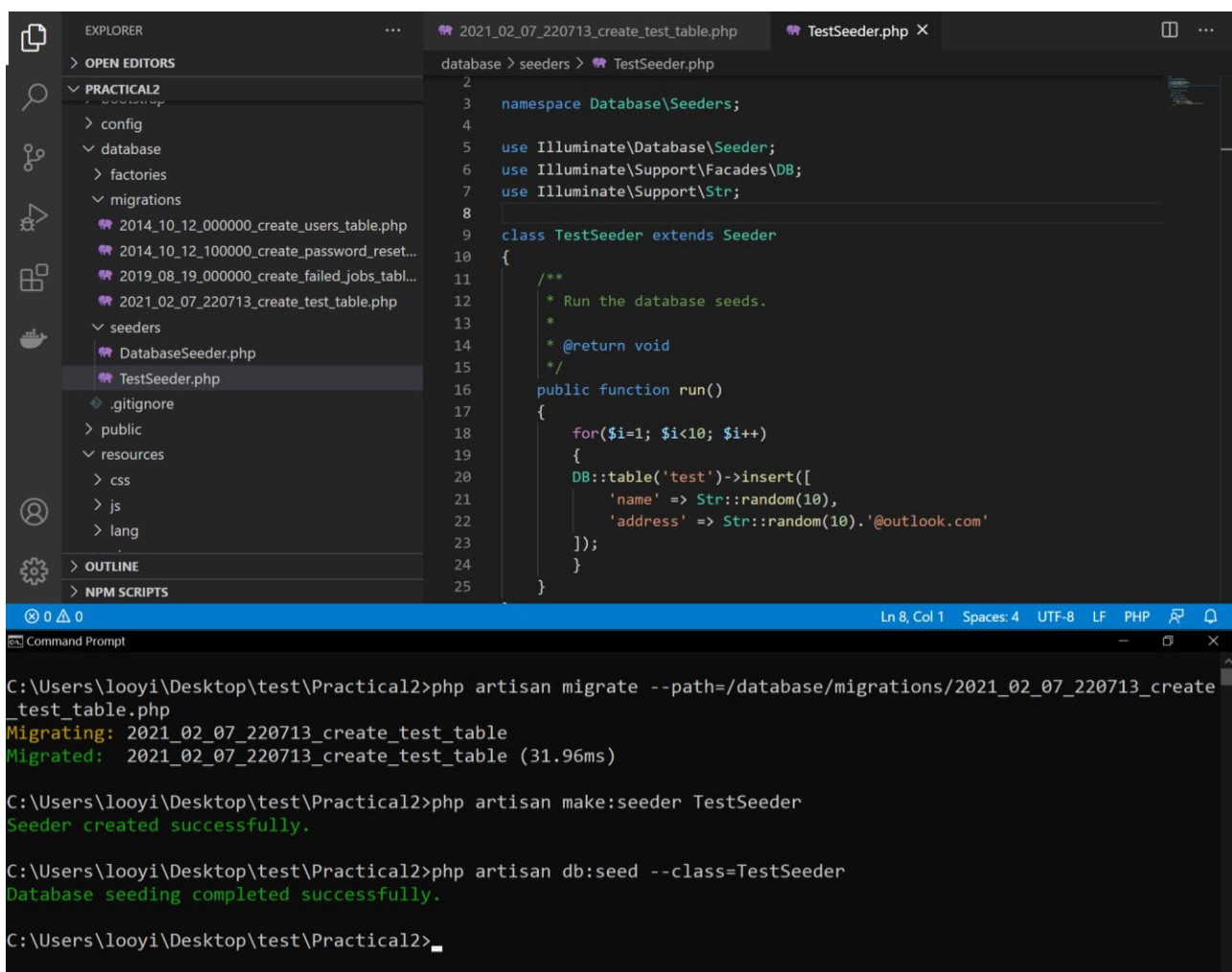
C:\Users\looyi\Desktop\test\Practical2>php artisan make:seeder TestSeeder
Seeder created successfully.

C:\Users\looyi\Desktop\test\Practical2>
```

Figure 21: Creating data seeding file with Artisan CLI.

The seeder file created is located in **Database\Seeders**. Within the seeder file, further define the import of database that the seeder file will interact with and dummy data that should be generated. After specification of data seeding, execute Artisan CLI to execute data seeding as shown in Figure 22.

php artisan make:seeder TestSeeder
php artisan db:seed --class=TestSeeder



```
2021_02_07_220713_create_test_table.php
TestSeeder.php

database > seeders > TestSeeder.php
2
3 namespace Database\Seeders;
4
5 use Illuminate\Database\Seeder;
6 use Illuminate\Support\Facades\DB;
7 use Illuminate\Support\Str;
8
9 class TestSeeder extends Seeder
10 {
11     /**
12      * Run the database seeds.
13      * @return void
14      */
15
16     public function run()
17     {
18         for($i=1; $i<10; $i++)
19         {
20             DB::table('test')->insert([
21                 'name' => Str::random(10),
22                 'address' => Str::random(10).'@outlook.com'
23             ]);
24         }
25     }
26 }
```

```
C:\Users\looyi\Desktop\test\Practical2>php artisan migrate --path=/database/migrations/2021_02_07_220713_create_test_table.php
Migrating: 2021_02_07_220713_create_test_table
Migrated: 2021_02_07_220713_create_test_table (31.96ms)

C:\Users\looyi\Desktop\test\Practical2>php artisan make:seeder TestSeeder
Seeder created successfully.

C:\Users\looyi\Desktop\test\Practical2>php artisan db:seed --class=TestSeeder
Database seeding completed successfully.

C:\Users\looyi\Desktop\test\Practical2>
```

Figure 22: Data seeding specification and execution.