**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).

# 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.

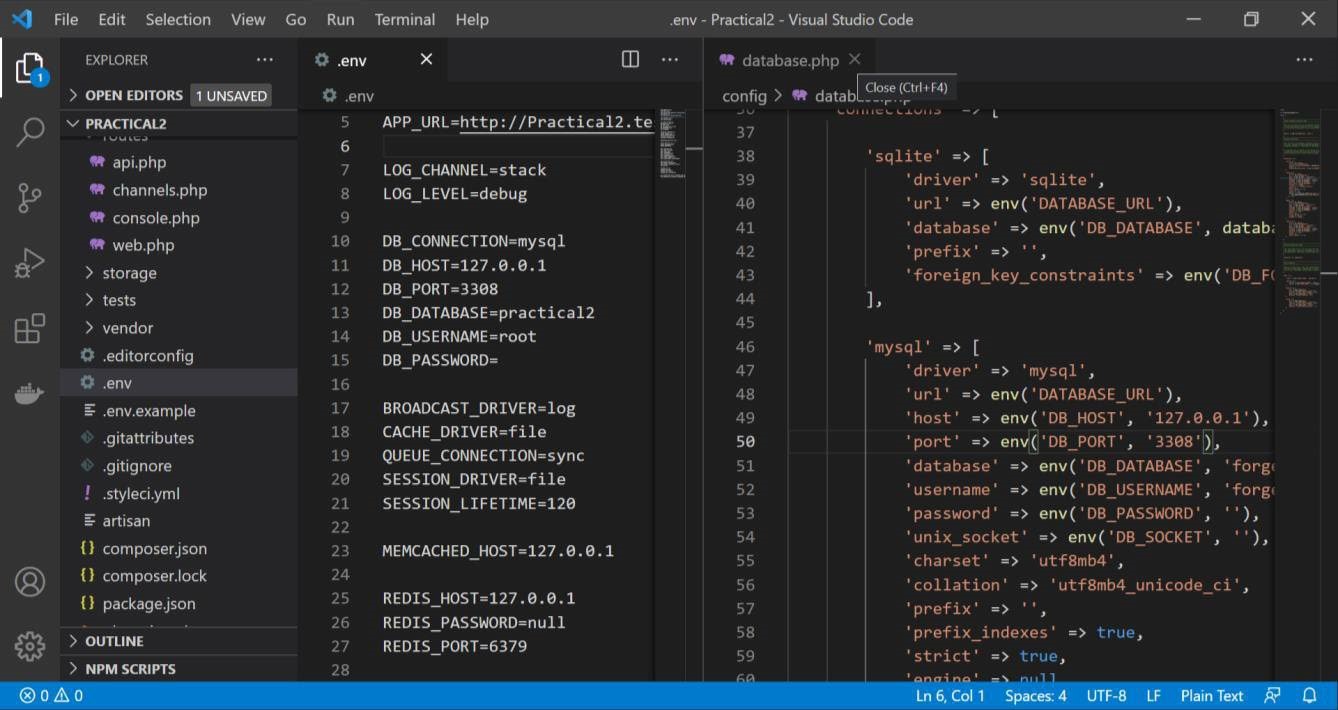


Figure 1: Database configuration file and lists for Laravel Web Application’s database.

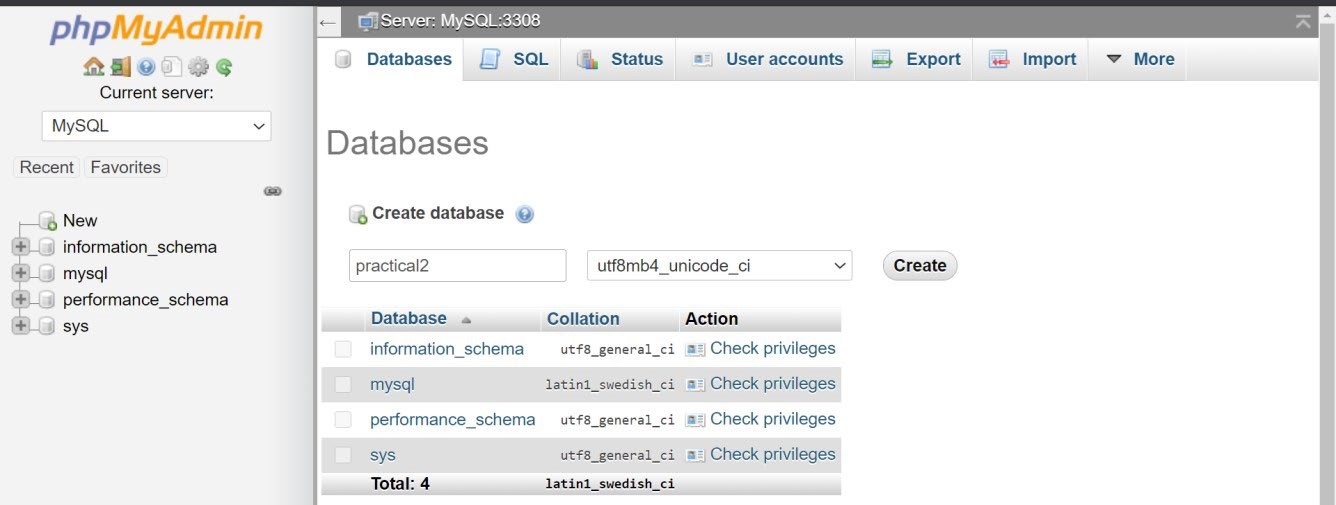


Figure 2: MySQL database to be used by Laravel Web Application.

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.

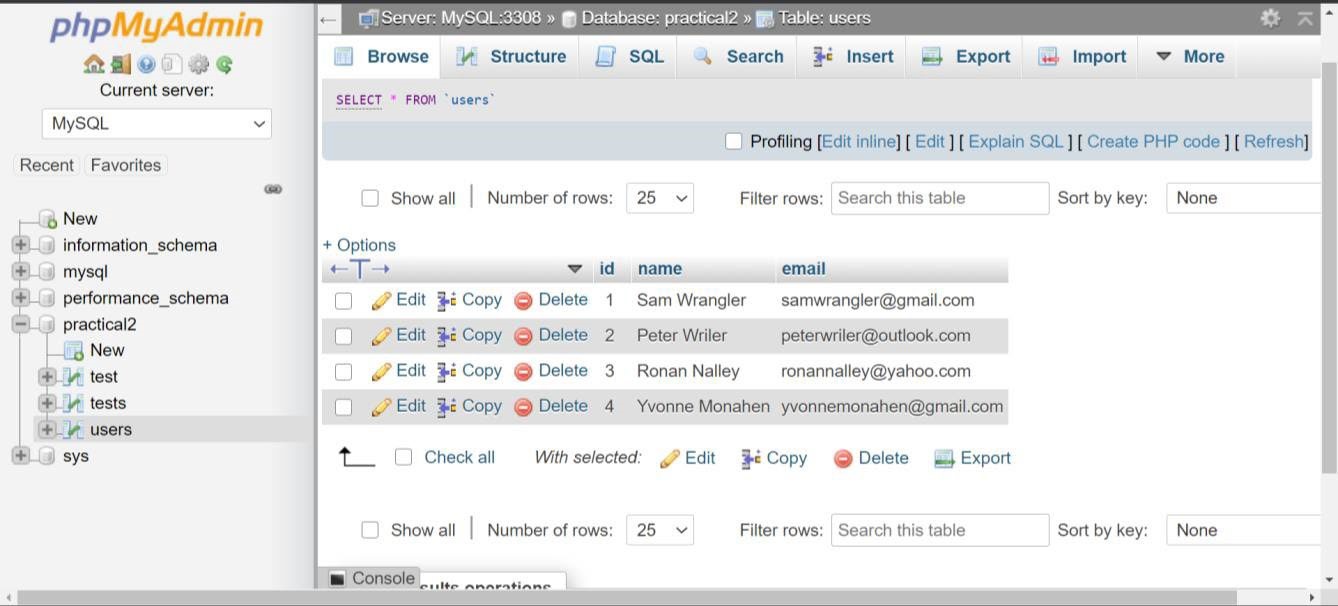


Figure 3: A users database table.

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.

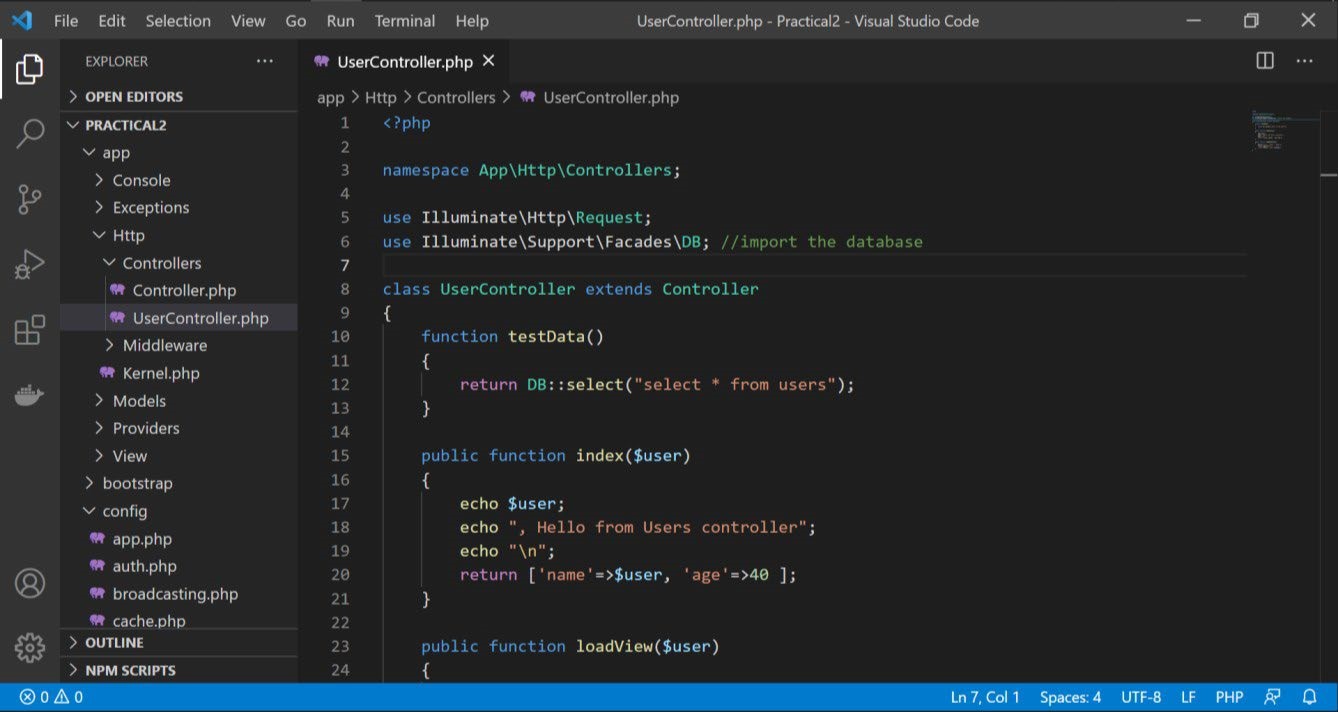


Figure 4: Raw SQL queries for data fetch using a controller.

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.

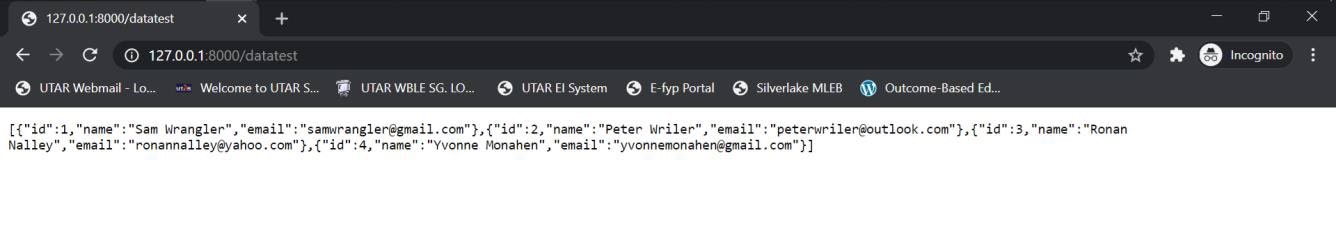


Figure 5: Fetch data from practical 2 database users table using controller.

# 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.

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

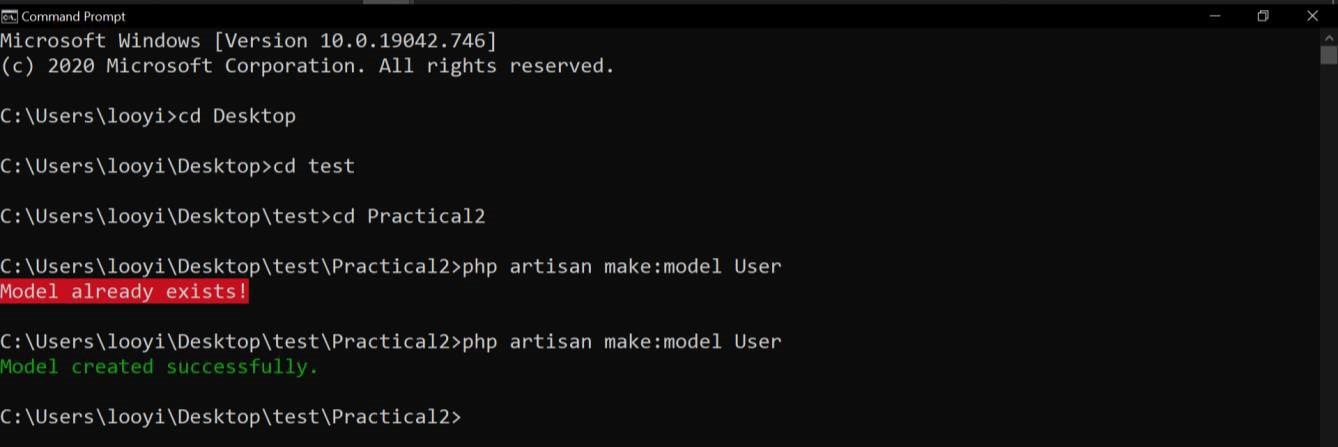


Figure 6: Using Artisan CLI to create User model.

1. 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.

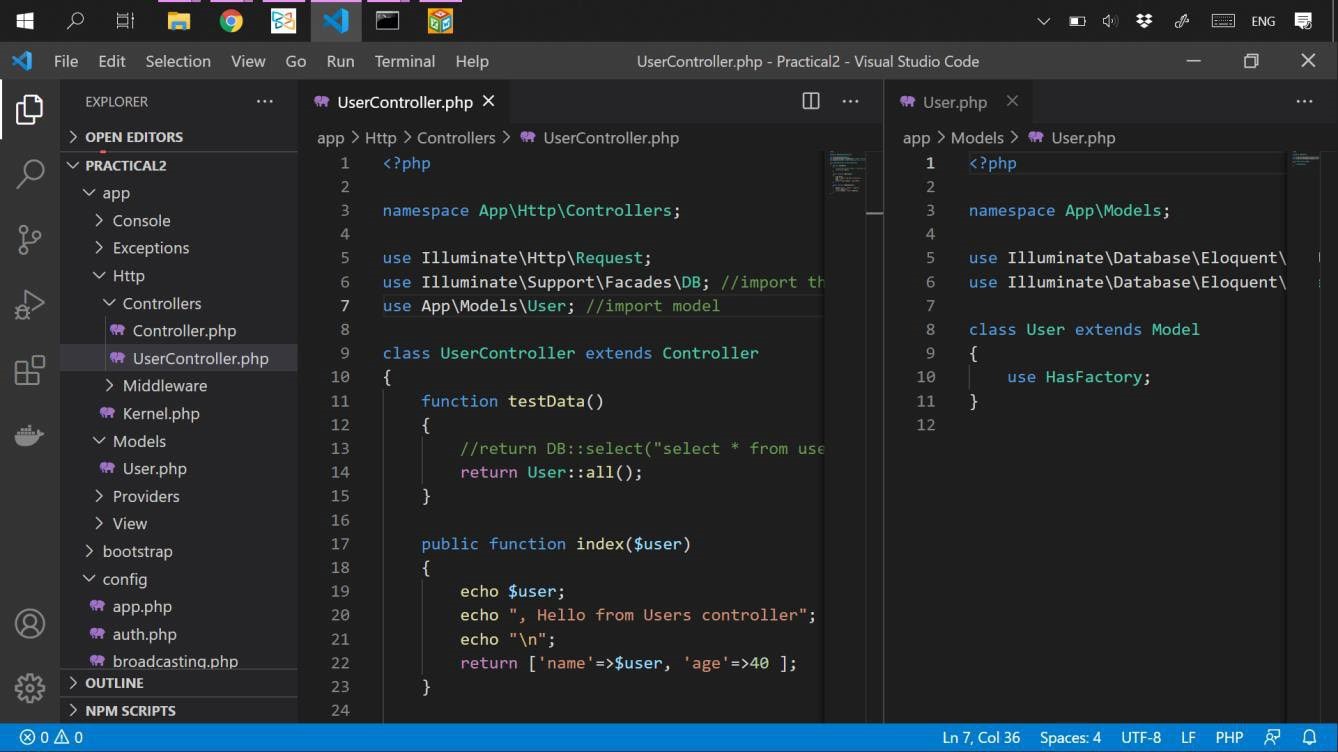


Figure 7: An echoing function in Users Controller.

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.

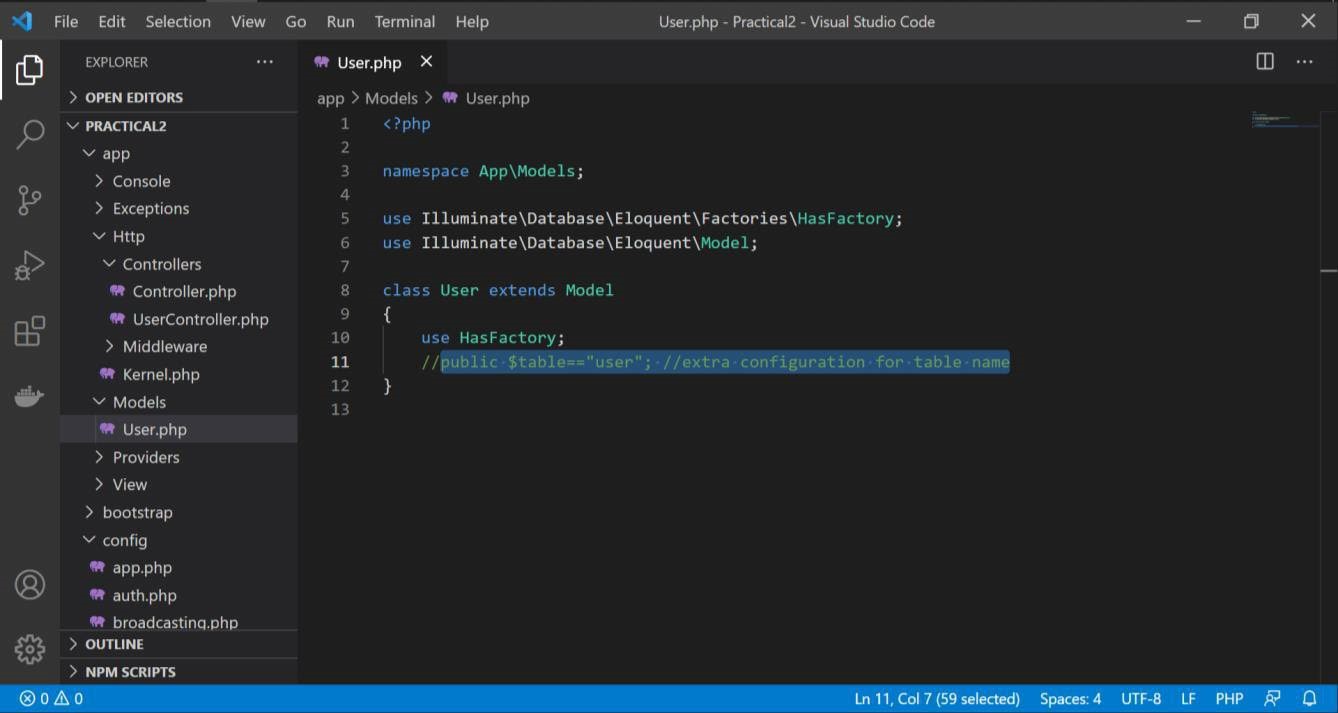


Figure 8: Extra configuration in model class for manual table name mapping.

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.

## 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.

A screen shot of a computer program

Description automatically generated

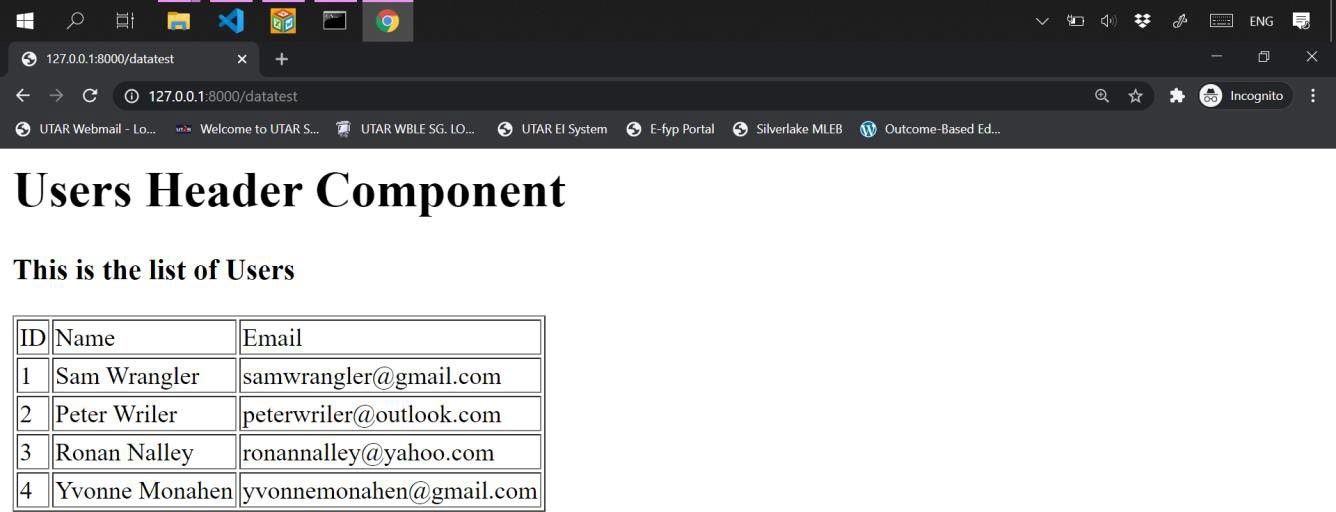
A screen shot of a computer screen

Description automatically generated

**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.



A screenshot of a computer program

Description automatically generated

## 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.

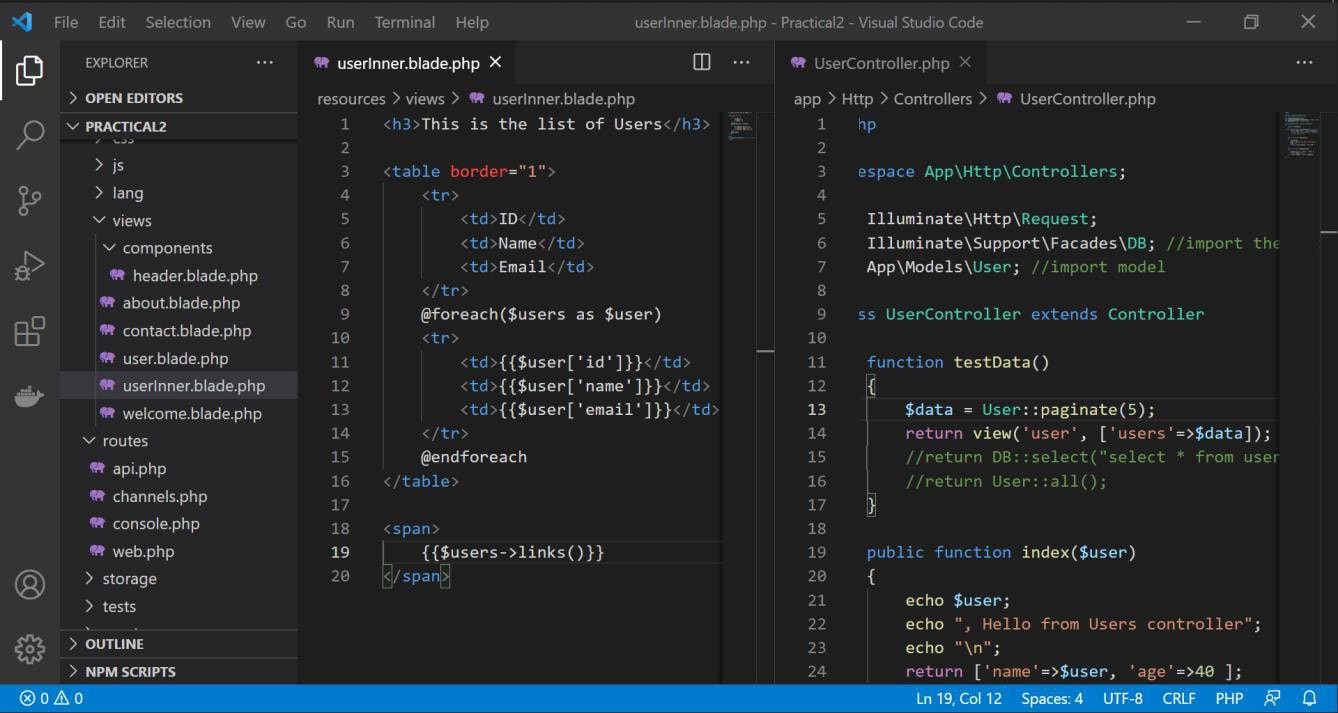


Figure 9: Applying pagination and displaying outputs in different pages.

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

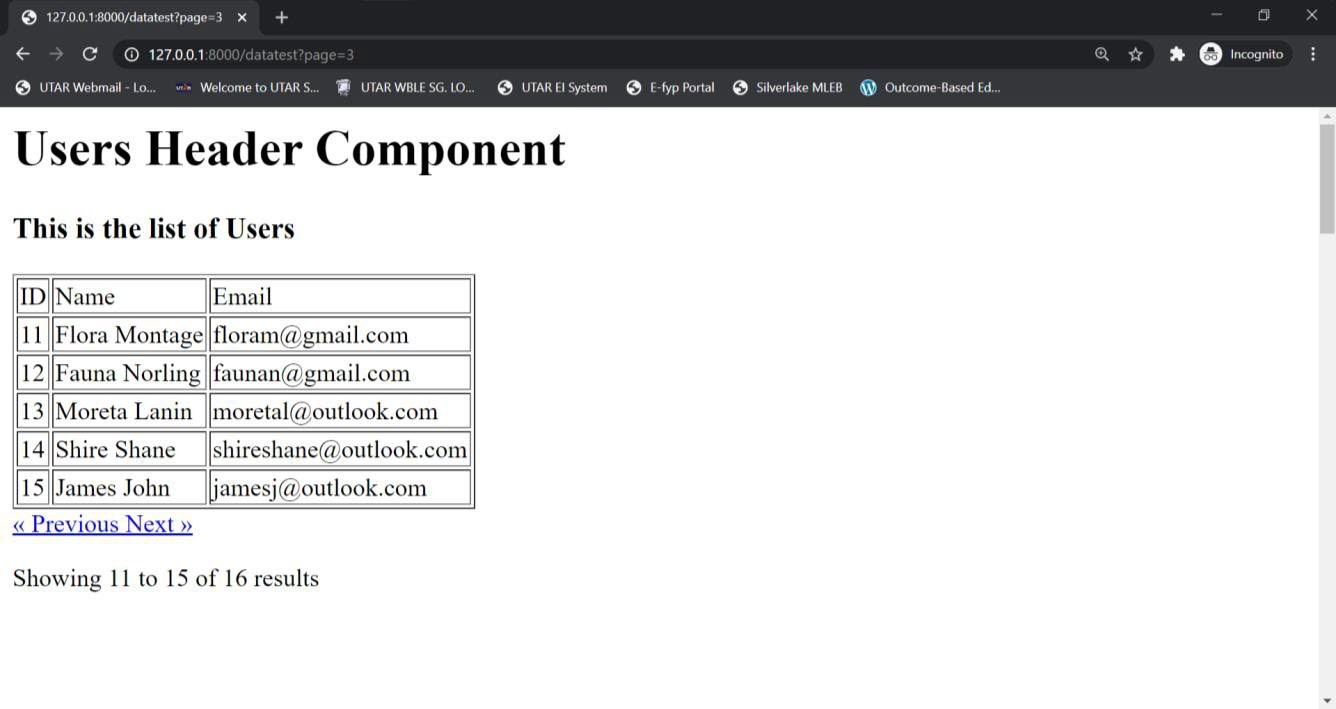


Figure 10: List of users with pagination.

|  |  |
| --- | --- |
| 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.

A screenshot of a computer program

Description automatically generated

## 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**.

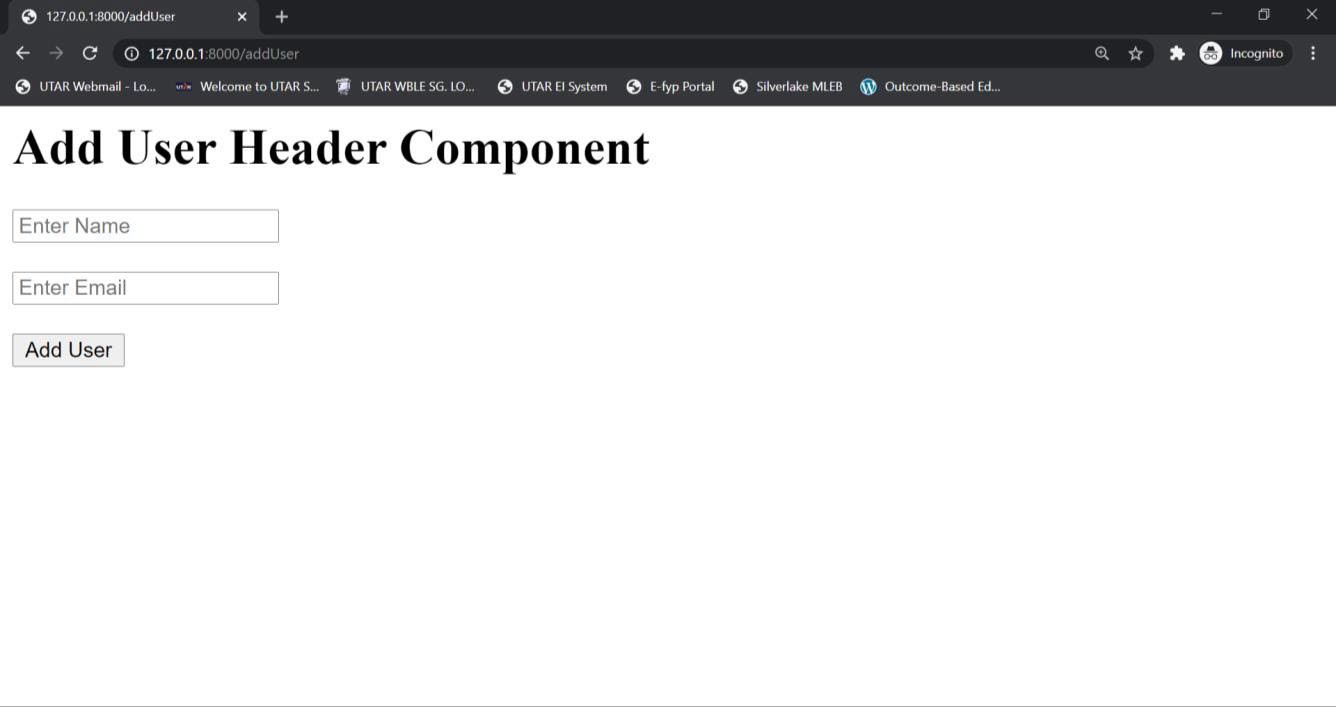


Figure 11: A simple form View for inputs.

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.

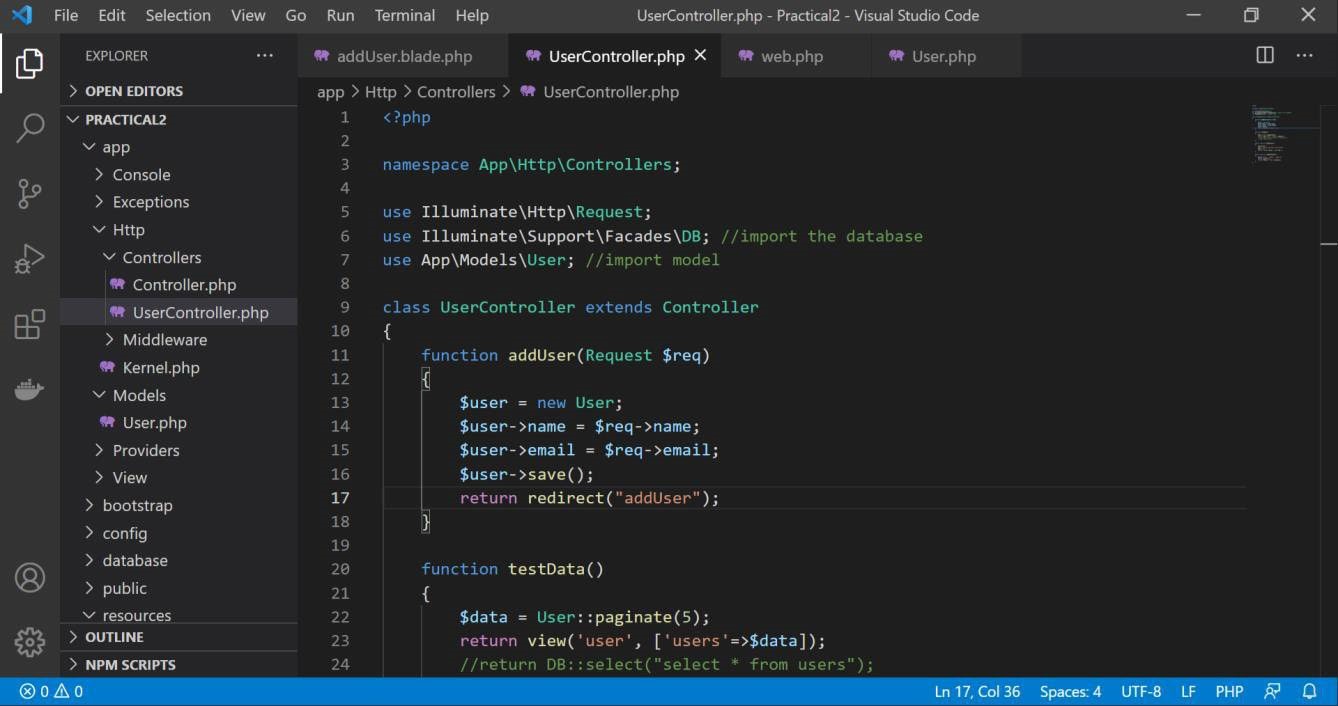
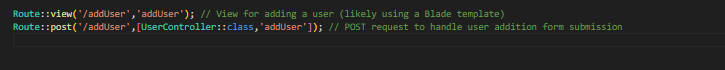


Figure 12: addUser function in UserController to process inputs.

A screen shot of a computer program

Description automatically generated



A screen shot of a computer program

Description automatically generated

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.

A computer screen shot of a program code

Description automatically generated

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.

## 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.

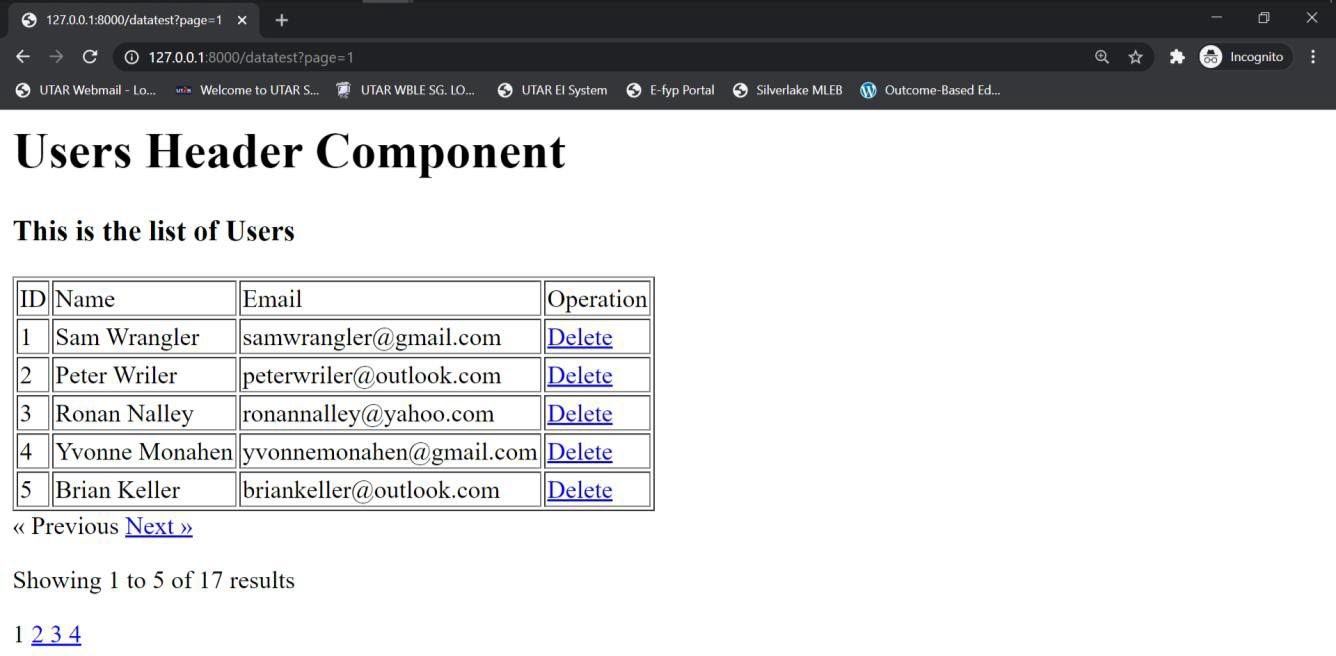


Figure 13: Delete function in userInner view for data deletion.

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

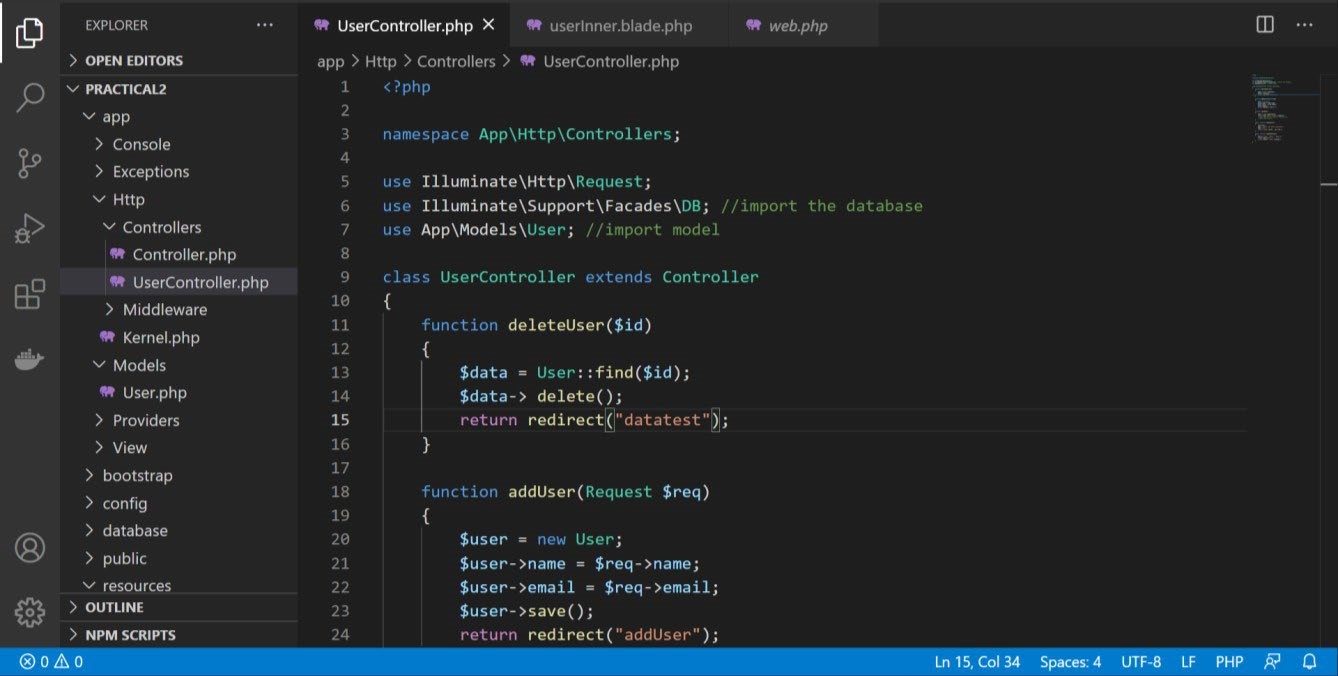


Figure 14: deleteUser function in UserController to process data deletion.

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

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.

## 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.

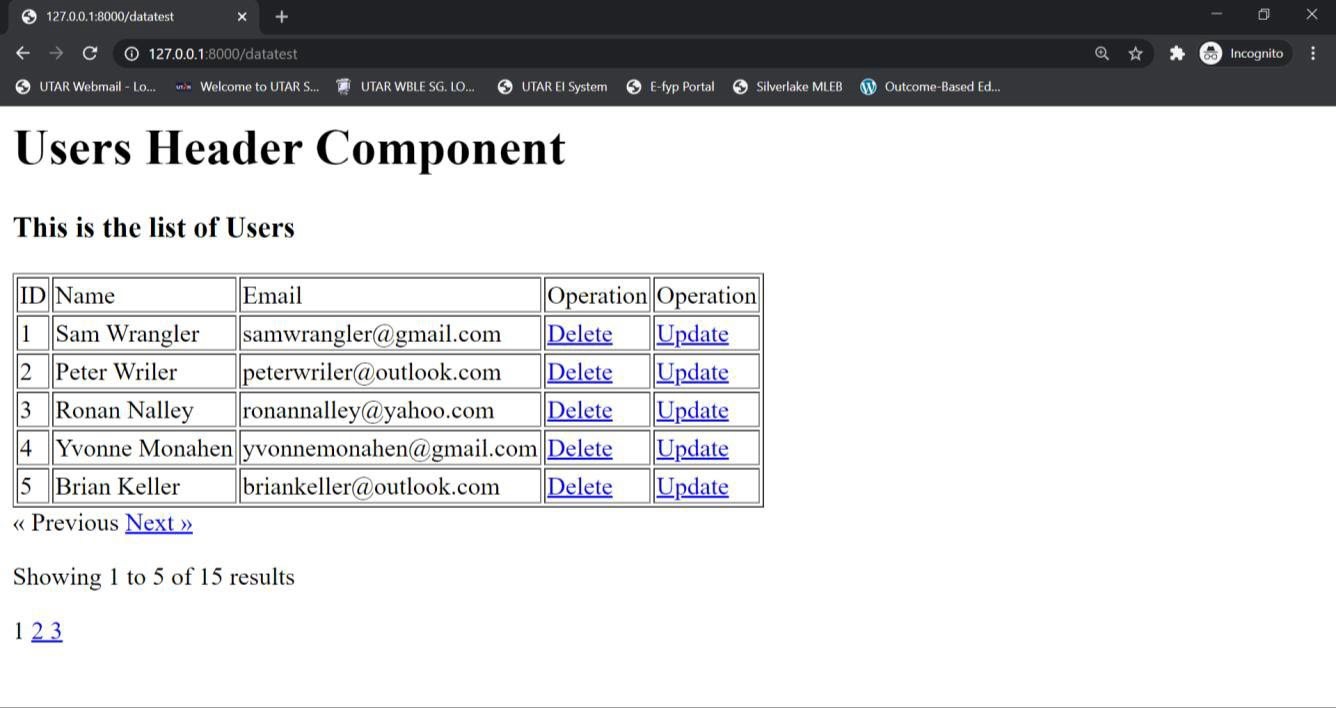


Figure 14: updateUser function in userInner view for data update.

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.

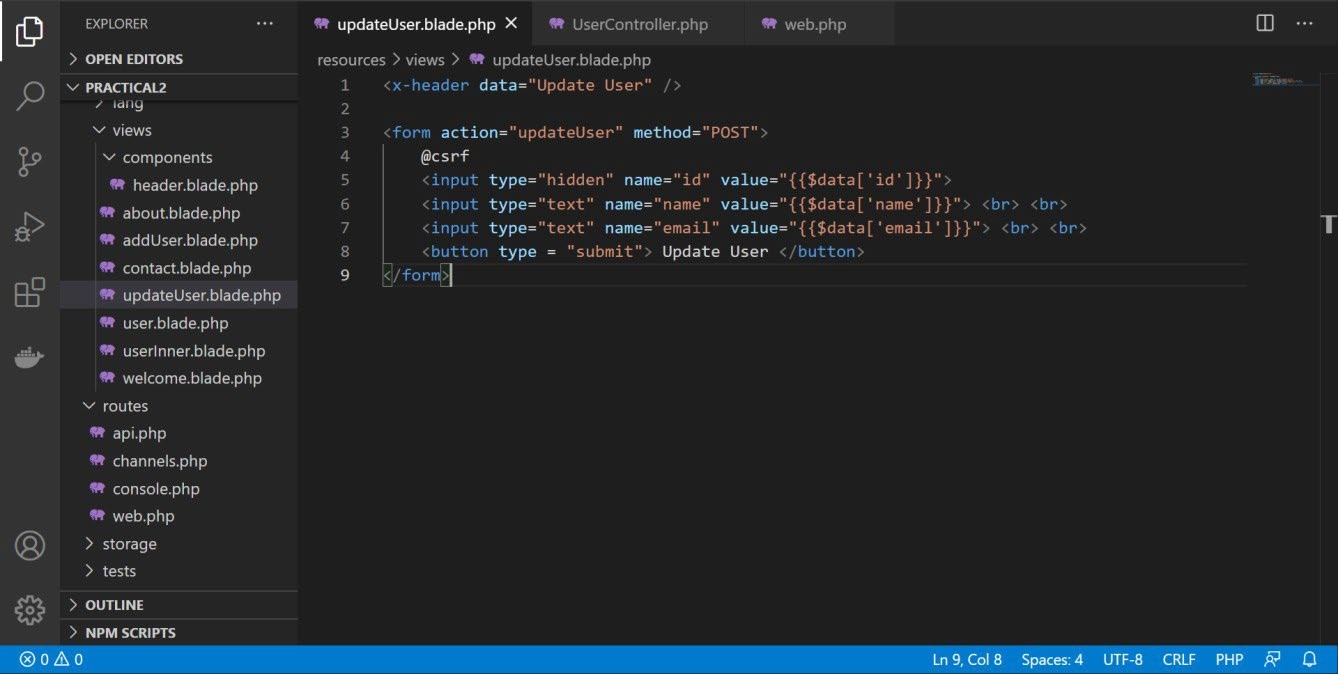


Figure 15: updateUser form view for data update.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

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.

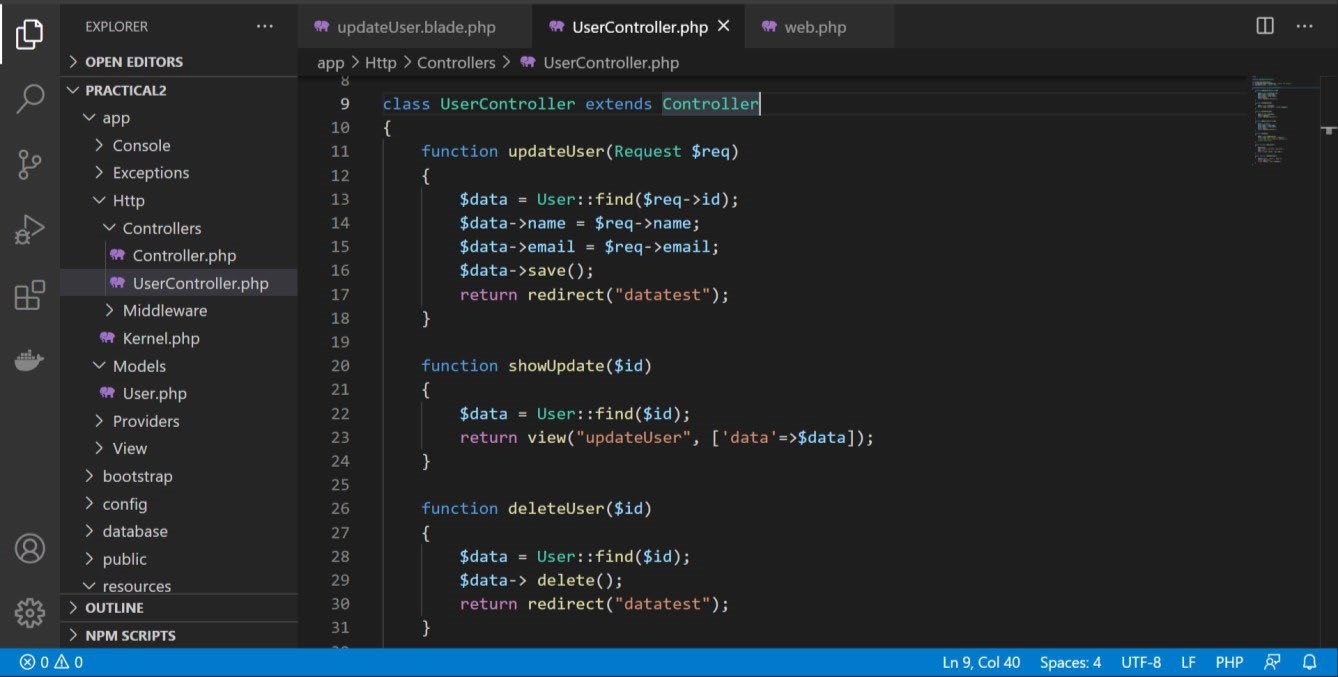


Figure 16: updateUser logics in UserController.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer screen

Description automatically generated

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

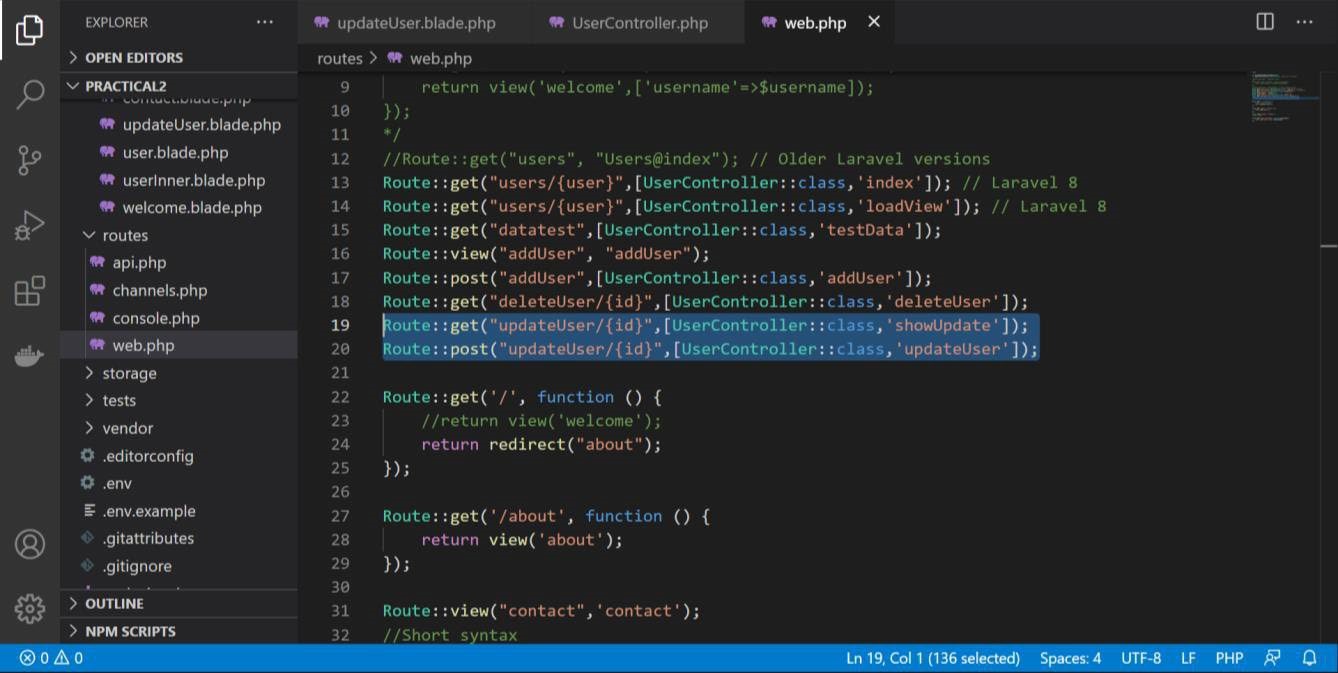


Figure 17: updateUser routes to UserController.



A screen shot of a computer program

Description automatically generated

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

# 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**

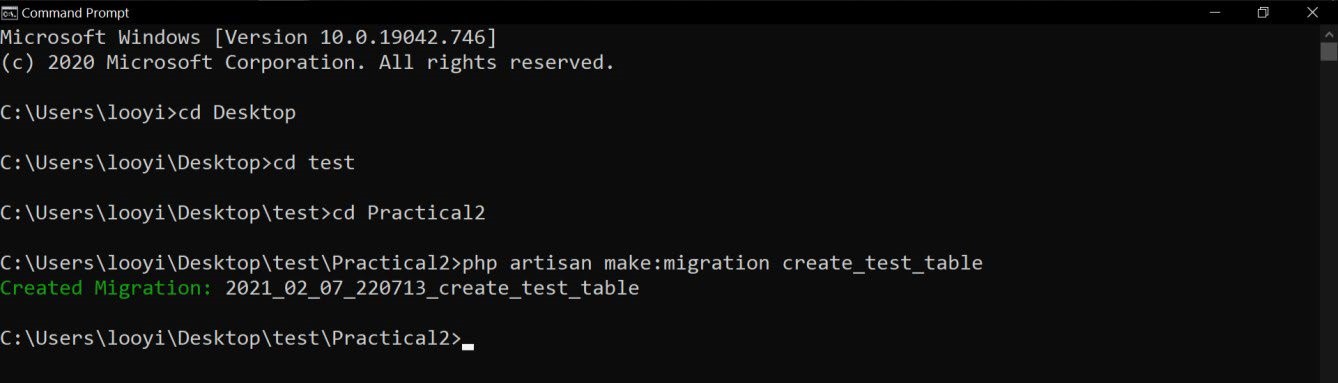


Figure 18: Data migration initiation using Artisan CLI.

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

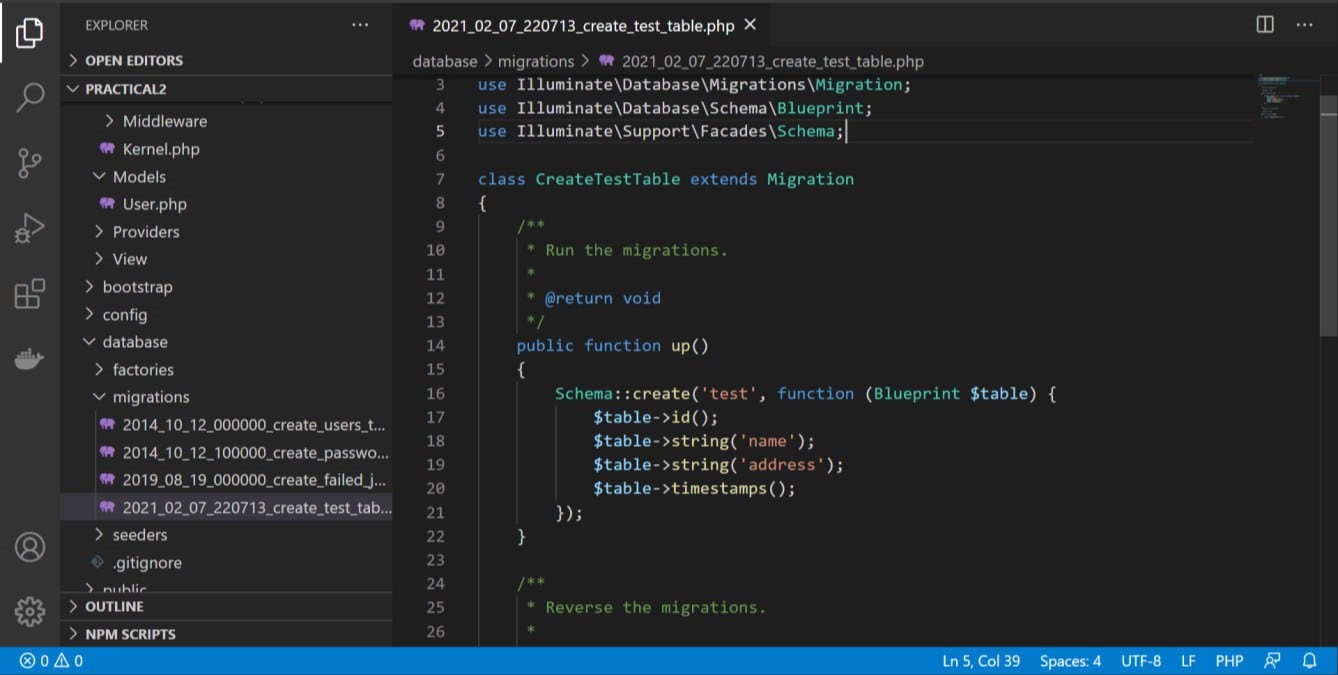


Figure 19: test database table structure.

A screen shot of a computer program

Description automatically generated

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**



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.

# 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.

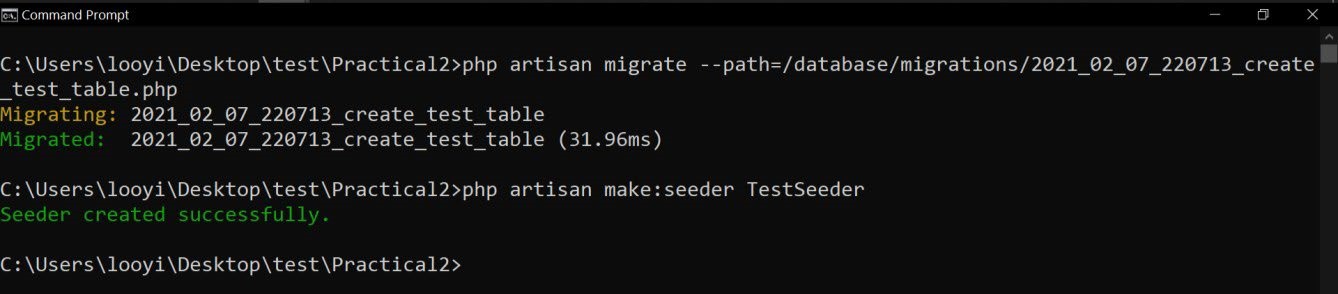


Figure 21: Creating data seeding file with Artisan CLI.

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**

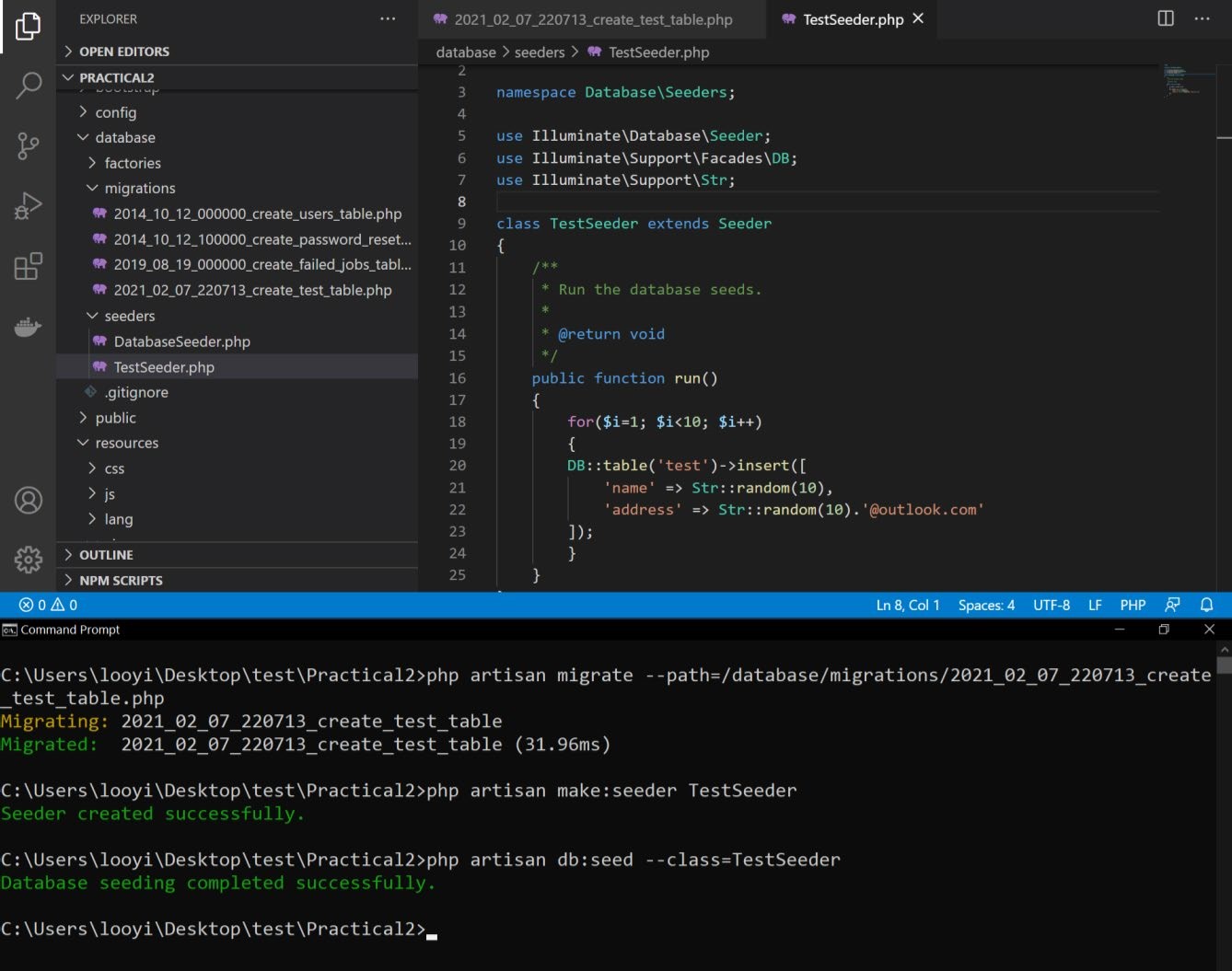


Figure 22: Data seeding specification and execution.

## Working with Database in Laravel

### Preparation

1. Make sure Wamp is started and no two instances of MySQL is started. Create a database in PhpMyAdmin. The database name can be the app name. Change the database name in the .env file. Check if all information is correct in .env and database.php.

|  |
| --- |
| // .env file  DB\_CONNECTION=mysql  DB\_HOST=127.0.0.1  DB\_PORT=3306  DB\_DATABASE=practical2  DB\_USERNAME=root  DB\_PASSWORD= |
| // config/database.php  'mysql' => [              'driver' => 'mysql',              'url' => env('DATABASE\_URL'),              'host' => env('DB\_HOST', '127.0.0.1'),              'port' => env('DB\_PORT', '3306'),              'database' => env('DB\_DATABASE', 'forge'),              'username' => env('DB\_USERNAME', 'forge'),              'password' => env('DB\_PASSWORD', ''),              'unix\_socket' => env('DB\_SOCKET', ''),              'charset' => 'utf8mb4',              'collation' => 'utf8mb4\_unicode\_ci',              'prefix' => '',              'prefix\_indexes' => true,              'strict' => true,              'engine' => null,              'options' => extension\_loaded('pdo\_mysql') ? array\_filter([                  PDO::MYSQL\_ATTR\_SSL\_CA => env('MYSQL\_ATTR\_SSL\_CA'),              ]) : [],          ], |

1. Directly interacting with database without using model and utilizing ORM.

|  |
| --- |
| public function index() {      return DB::select('select \* from users');  } |

1. To use ORM, recommended way, you need a model and a table in the database. **The table name must be the plural of the model’s name.** For example, the plural of user is users. To create a model, use this command. **Ensure the columns in the database table are the same as the model’s attributes.** You can add test data manually in PhpMyAdmin.

|  |
| --- |
| php artisan make:model User |

### CRUD (Read)

1. CRUD(R): To read all from table using ORM, first, use the “Model::all” static function from the model. The “Model::all” static function is there by default in any model.

|  |
| --- |
| public function loadAllData() {      return User::all();  } |

1. CRUD(R): To paginate a specific number of data from table using ORM, use Model::paginate(5).

|  |
| --- |
| public function loadData() {    $users = User::paginate(5);      return view('/users', ['users'=>$users]);  } |

1. CRUD(R): An example of how to display paginated data in a table.

|  |
| --- |
| <table border="1">        <thead>              <tr>                  <th>id</th>                  <th>name</th>                  <th>email</th>              </tr>          </thead>          <tbody>              @foreach ($users as $user)                  <tr>                      <td>{{ $user['id'] }}</td>                      <td>{{ $user['name'] }}</td>                      <td>{{ $user['email'] }}</td>                  </tr>              @endforeach          </tbody>  </table>  <span>       {{ $users->links() }}  </span> |

### CRUD(Create)

1. For the create operation, first, create a create page that allows the user to enter the data of the new row that can be accessed through the /addUser route. The action is the same route, but using the post method. Remember to use @csrf.

|  |
| --- |
| <form action="/addUser" method="post">      @csrf      <input type="text" name="id" placeholder="Enter id"><br><br>      <input type="text" name="name" placeholder="Enter name"><br><br>      <input type="text" name="email" placeholder="Enter email"><br><br>      <button type="submit">Add User</button>  </form> |

Add the route with two different methods in the web.php.

|  |
| --- |
| Route::view('/addUser', 'addUser'); |

|  |
| --- |
| Route::post('/addUser', [UsersController::class, "addUser"]); |

1. Then, create a function in controller to process the request and saves the data when the user press the button and redirects to display all data page. The important thing is to have these two things: addUser.blade.php, and addUser function in controller.

|  |
| --- |
| public function addUser(Request $request)  {      $user = new User();      $user->id = $request->id;      $user->name = $request->name;      $user->email = $request->email;      $user->save();      return redirect('/users');  } |

### CRUD Update

1. Setup update user form page, controller function that shows the update form page with user where the id is from URL, controller function that updates the user.

|  |
| --- |
| // 1. updateUser.blade.php  // when the update user button is pressed, the action goes to /updateUser route and use post method is executed.  <form action="/updateUser" method="post">      @csrf      <input type="hidden" name="id" value="{{$user['id']}}">      <input type="text" name="name" value="{{$user['name']}}"> <br><br>      <input type="text" name="email" value="{{$user['email']}}"> <br><br>      <button type="submit">Update User</button>  </form> |

|  |
| --- |
| // 2. Controller function that shows the updateUser view.  // It should take in an id from the route and displays the user info in the view.  public function showUpdatePage($id)  {      $user = User::find($id);      return view('updateUser', ['user'=>$user]);  } |

|  |
| --- |
| // 3. Controller function that updates the user.  // Must take in request in the form of an user and SPECIFY Request as the data type, to perform the update.  public function updateUser(Request $request) {      $userToUpdate = User::find($request->id);      $userToUpdate->name = $request->name;      $userToUpdate->email = $request->email;      $userToUpdate->save();      return redirect('/users');  } |

1. Setup the route for these the update functions in the controller.

|  |
| --- |
| Route::get('/updateUser/{id}', [UsersController::class, "showUpdatePage"]);  Route::post('/updateUser', [UsersController::class, "updateUser"]); |

1. Example of table

|  |
| --- |
| <table border="1">          <thead>              <tr>                  <th>id</th>                  <th>name</th>                  <th>email</th>                  <th>Operation</th>                  <th>Operation</th>              </tr>          </thead>          <tbody>              @foreach ($users as $user)                  <tr>                      <td>{{ $user['id'] }}</td>                      <td>{{ $user['name'] }}</td>                      <td>{{ $user['email'] }}</td>                      <td><a href="/updateUser/{{$user['id']}}">Update</a></td>                      <td><a href="/deleteUser/{{$user['id']}}">Delete</a></td>                  </tr>              @endforeach          </tbody>      </table> |

### CRUD Delete

|  |
| --- |
| public function deleteUser($id) {      $userToDelete = User::find($id);      $userToDelete->delete();      return redirect('/users');  } |

1. Setup the delete function in controller. It should take in an id and deletes the user corresponding to that id.
2. Specify the route of the function.

|  |
| --- |
| Route::get('/deleteUser/{id}', [UsersController::class, "deleteUser"]); |

### Migrations

1. Migrations are test database tables that developers should work with instead of touching the real database. First step of creating migrations is to create the migration class that specifies the database table and its attributes.

The command to create a migration class is as follows. A file like 2024\_02\_24\_021632\_create\_test\_table.php will be created in database/migrations.

|  |
| --- |
| php artisan make:migration create\_test\_table |

Edit the migration file to the table name that you want, and then add the add the table attributes in the schema create function.

|  |
| --- |
| public function up()      {          Schema::create('testMigrateTable', function (Blueprint $table) {              $table->id('id');              $table->string('name');              $table->string('email');          });  }      public function down()      {          Schema::dropIfExists('testMigrateTable');      } |

1. Then, you need to execute the migration class to create the actual table in the database.

The command to execute the migration class is as follows. The example of migration\_name is 2024\_02\_24\_021632\_create\_test\_table.php.

|  |
| --- |
| php artisan migrate --path=/database/migrations/migration\_name.php |

1. Finally, you need to seed the database with data. To do so, create a class that specifies the data to seed it.

The command to create a seeder class is as follows:

|  |
| --- |
| php artisan make:seeder TestSeeder |

A TestSeeder class will be created located in database/seeders. Edit the run function to specify how you want to seed the data.

|  |
| --- |
| public function run()      {          for($i=0; $i<10; $i++)          {              DB::table('testMigrateTable')->insert([                  'id' => $i+1,                  'name' => Str::random(10),                  'email' => Str::random(10).'gmail.com'              ]);          }      } |

To execute the TestSeeder class run function that seeds the migration database, execute the command as follows.

|  |
| --- |
| php artisan db:seed --class=TestSeeder |