

Practical 3 : Working with Databases and Object-Relational Mapping

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 interface with two open files:

- .env**: Contains environment variables for the application, including APP_URL, LOG_CHANNEL, LOG_LEVEL, DB_CONNECTION, DB_HOST, DB_PORT, DB_DATABASE, DB_USERNAME, DB_PASSWORD, BROADCAST_DRIVER, CACHE_DRIVER, QUEUE_CONNECTION, SESSION_DRIVER, SESSION_LIFETIME, MEMCACHED_HOST, REDIS_HOST, REDIS_PASSWORD, and REDIS_PORT.
- database.php**: A configuration file for the database connection, defining two drivers: 'sqlite' and 'mysql'. The 'sqlite' driver uses the URL from the .env file. The 'mysql' driver uses the DB_* variables from the .env file.

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

The screenshot shows the phpMyAdmin interface for MySQL 3308. The left sidebar shows the current server (MySQL) and recent databases: information_schema, mysql, performance_schema, and sys. The main area is titled "Databases" and shows a "Create database" form with "practical2" entered in the "Database" field and "utf8mb4_unicode_ci" selected in the "Collation" dropdown. Below the form is a table of existing databases with their collations and actions:

Database	Collation	Action
information_schema	utf8_general_ci	Check privileges
mysql	latin1_swedish_ci	Check privileges
performance_schema	utf8_general_ci	Check privileges
sys	utf8_general_ci	Check privileges
Total: 4		

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

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.

The screenshot shows the phpMyAdmin interface for a MySQL database named 'practical2'. The left sidebar lists databases: 'information_schema', 'mysql', 'performance_schema', 'practical2', 'test', 'tests', and 'sys'. The 'users' database is selected. The main area displays a table named 'users' with the following data:

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

Below the table are buttons for 'Check all', 'With selected:', and 'Edit' (with options for Copy, Delete, Export).

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.

The screenshot shows the Visual Studio Code editor with the file 'UserController.php' open. The code defines a UserController class extending the Controller class. It contains two methods: testData() and index().

```
<?php  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\DB; //import the database  
  
class UserController extends Controller  
{  
    function testData()  
    {  
        return DB::select("select * from users");  
    }  
  
    public function index($user)  
    {  
        echo $user;  
        echo ", Hello from Users controller";  
        echo "\n";  
        return [ 'name'=>$user, 'age'=>40 ];  
    }  
    public function loadView($user)  
    {  
        //  
    }  
}
```

The status bar at the bottom indicates the code has 24 lines, 0 errors, and is saved in PHP.

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.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

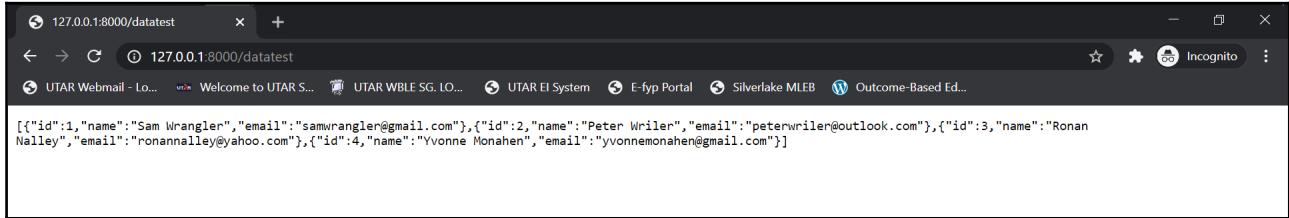


Figure 5: Fetch data from practical2 database users table using controller.

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.

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

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following command history:

```
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>
```

The command "php artisan make:model User" was run twice. The first attempt failed because the model already existed. The second attempt succeeded, as indicated by the green text "Model created successfully."

Figure 6: Using Artisan CLI to create User model.

- 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.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

The screenshot shows the Visual Studio Code interface with two code editors open. The left editor contains the file `UserController.php` from the `Http\Controllers` directory, which defines a controller with an `index` method that echoes user information. The right editor contains the file `User.php` from the `Models` directory, which defines a model class `User` extending `Model`. Both files use Laravel's Eloquent ORM syntax.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB; //import this
use App\Models\User; //import model

class UserController extends Controller
{
    function testData()
    {
        //return DB::select("select * from user");
        return User::all();
    }

    public function index($user)
    {
        echo $user;
        echo ", Hello from Users controller";
        echo "\n";
        return ['name'=>$user, 'age'=>40 ];
    }
}
```

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class User extends Model
{
    use HasFactory;
    //public $table=="user"; //extra configuration for table name
}
```

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 in the model class as shown in Figure 8.

The screenshot shows the Visual Studio Code interface with the file `User.php` from the `Models` directory open. The code includes a comment indicating a manual table name mapping: `//public $table=="user"; //extra configuration for table name`.

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    use HasFactory;
    //public $table=="user"; //extra configuration for table name
}
```

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

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.

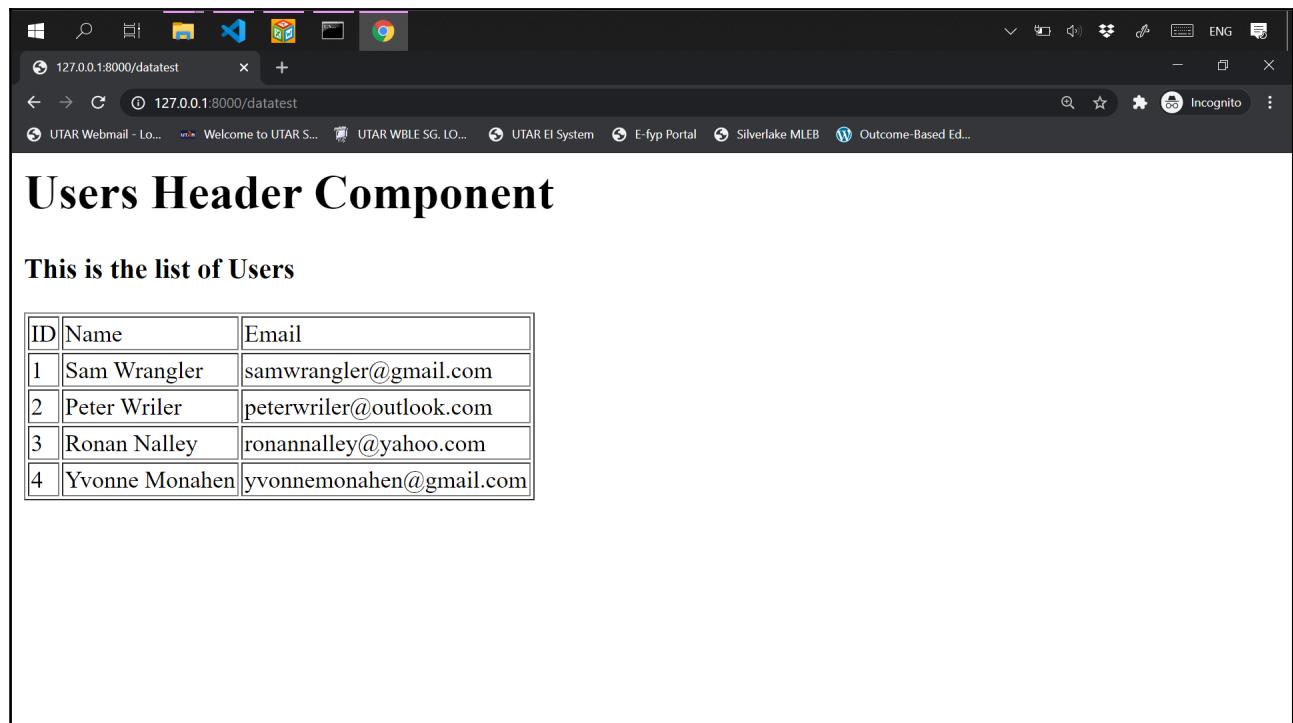
Within the controller, we know that “User::all()” contains all data from Users table. Previously, we used controller to return a view and passed some data to the view for output.

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 Microsoft Edge browser window with the URL "127.0.0.1:8000/datatest". The title bar says "127.0.0.1:8000/datatest". The page content is titled "Users Header Component". Below it, a heading "This is the list of Users" is displayed. A table follows, with columns labeled "ID", "Name", and "Email". The table contains four rows of data:

ID	Name	Email
1	Sam Wrangler	samwrangler@gmail.com
2	Peter Wriler	peterwriler@outlook.com
3	Ronan Nalley	ronannalley@yahoo.com
4	Yvonne Monahen	yvonnenmonahen@gmail.com

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.

The screenshot shows the Visual Studio Code interface with two tabs open. The left tab, titled 'userInner.blade.php', contains Blade template code for displaying a list of users. The right tab, titled 'UserController.php', contains PHP code for the UserController, specifically defining the paginate method and the index action which returns a view with user data.

```
userInner.blade.php
resources > views > userInner.blade.php
1  <h3>This is the list of Users</h3>
2
3  <table border="1">
4      <tr>
5          <td>ID</td>
6          <td>Name</td>
7          <td>Email</td>
8      </tr>
9      @foreach($users as $user)
10         <tr>
11             <td>{{$user['id']}}</td>
12             <td>{{$user['name']}}</td>
13             <td>{{$user['email']}}</td>
14         </tr>
15     @endforeach
16     </table>
17
18     <span>
19         {{$users->links()}}
20     </span>
```

```
UserController.php
app > Http > Controllers > UserController.php
1  hp
2
3  espace App\Http\Controllers;
4
5  Illuminate\Http\Request;
6  Illuminate\Support\Facades\DB; //import the
7  App\Models\User; //import model
8
9  ss UserController extends Controller
10
11     function testData()
12     {
13         $data = User::paginate(5);
14         return view('user', ['users'=>$data]);
15         //return DB::select("select * from user");
16         //return User::all();
17     }
18
19     public function index($user)
20     {
21         echo $user;
22         echo ", Hello from Users controller";
23         echo "\n";
24         return ['name'=>$user, 'age'=>40];
```

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.

The screenshot shows a web browser window displaying a list of users. The title bar indicates the URL is 127.0.0.1:8000/datatest?page=3. The page content includes a header 'Users Header Component', a sub-header 'This is the list of Users', and a table listing 16 users. At the bottom, there are navigation links '« Previous' and 'Next »'. Below the table, a message says 'Showing 11 to 15 of 16 results'.

ID	Name	Email
11	Flora Montage	floram@gmail.com
12	Fauna Norling	faunan@gmail.com
13	Moreta Lanin	moretal@outlook.com
14	Shire Shane	shireshane@outlook.com
15	James John	jamesj@outlook.com

Figure 10: List of users with pagination.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

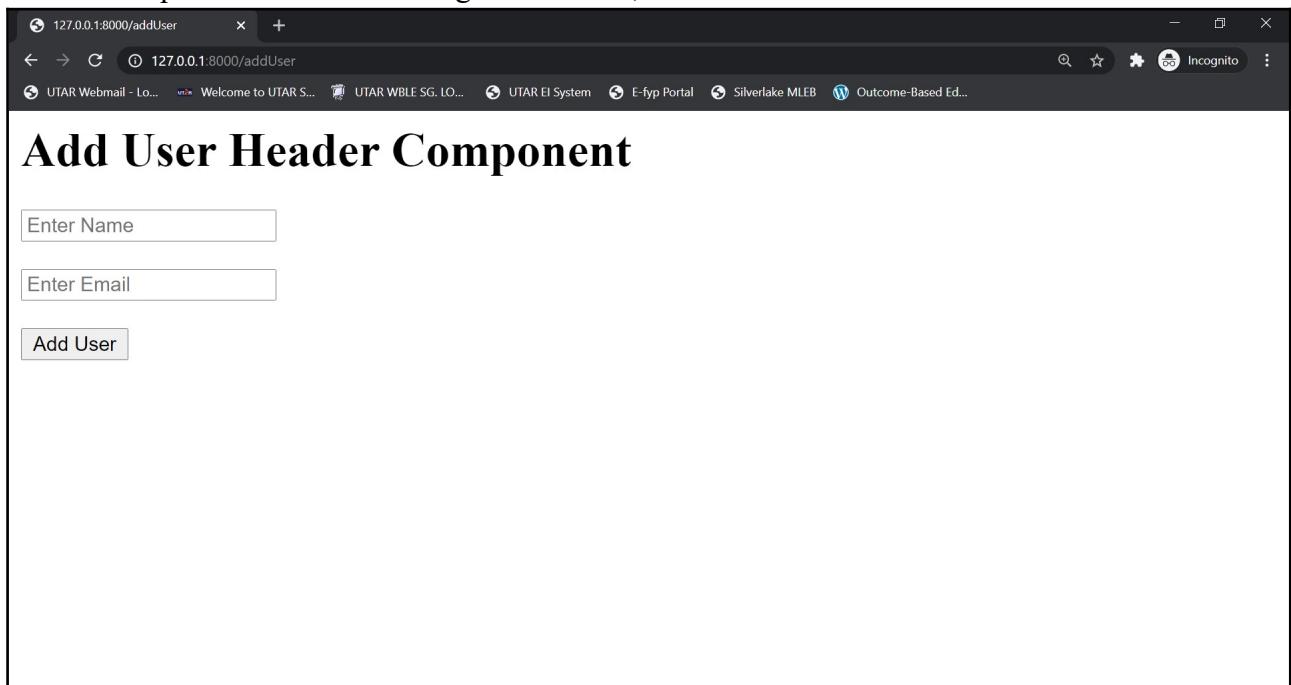
**There is a mishandling of CSS within Laravel framework. In order to fix the mishandling, include following scripts to the view file.

```
<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.

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.

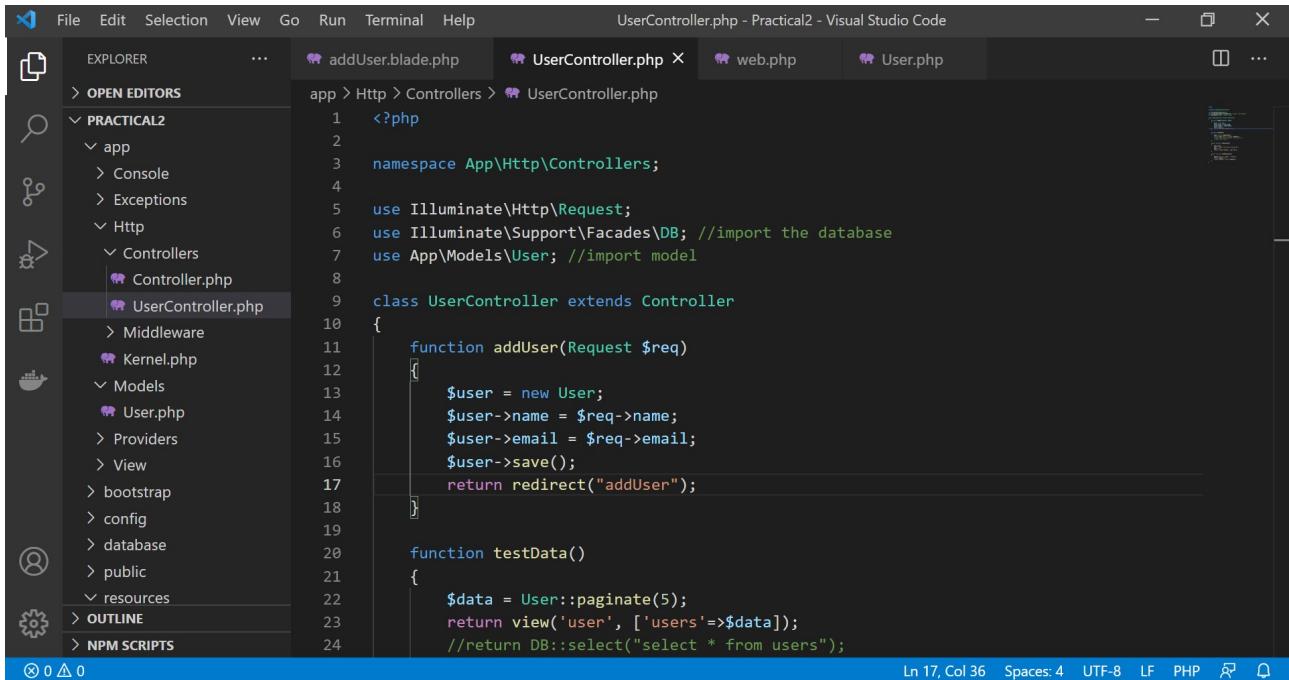


The screenshot shows a web browser window with the URL `127.0.0.1:8000/addUser`. The title bar says "Add User Header Component". The page contains two input fields: "Enter Name" and "Enter Email", followed by a "Add User" button. The browser's address bar also shows the URL `127.0.0.1:8000/addUser`.

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.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** UserController.php - Practical2 - Visual Studio Code.
- Explorer:** Shows the project structure under 'PRACTICAL2': app, Http, Controllers, Kernel.php, Models, User.php, Providers, View, bootstrap, config, database, public, resources, OUTLINE, NPM SCRIPTS.
- Editor:** Displays the UserController.php file with the following code:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB; //import the database
use App\Models\User; //import model

class UserController extends Controller
{
    function addUser(Request $req)
    {
        $user = new User;
        $user->name = $req->name;
        $user->email = $req->email;
        $user->save();
        return redirect("addUser");
    }

    function testData()
    {
        $data = User::paginate(5);
        return view('user', ['users'=>$data]);
        //return DB::select("select * from users");
    }
}
```

Bottom status bar: Ln 17, Col 36, Spaces: 4, UTF-8, LF, PHP, ⚡, ⚡.

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.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.

The screenshot shows a web browser window with the URL `127.0.0.1:8000/datatest?page=1`. The page title is "Users Header Component". Below it, a heading says "This is the list of Users". A table lists five users with columns for ID, Name, Email, and Operation (containing a "Delete" link). Below the table are navigation links "« Previous" and "Next »", and a page number indicator "1 2 3 4".

ID	Name	Email	Operation
1	Sam Wrangler	samwrangler@gmail.com	Delete
2	Peter Wriler	peterwriler@outlook.com	Delete
3	Ronan Nalley	ronannalley@yahoo.com	Delete
4	Yvonne Monahen	yvonnemonahen@gmail.com	Delete
5	Brian Keller	briankeller@outlook.com	Delete

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.

The screenshot shows a code editor with the file `UserController.php` open. The code defines a `UserController` class extending `Controller`. It contains two methods: `deleteUser` and `addUser`. The `deleteUser` method finds a user by ID and deletes it, then redirects to the "datatest" page. The `addUser` method creates a new `User` object, sets its name and email from the request, saves it to the database, and then redirects to the "addUser" page.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB; //import the database
use App\Models\User; //import model
class UserController extends Controller
{
    function deleteUser($id)
    {
        $data = User::find($id);
        $data-> delete();
        return redirect(["datatest"]);
    }
    function addUser(Request $req)
    {
        $user = new User;
        $user->name = $req->name;
        $user->email = $req->email;
        $user->save();
        return redirect("addUser");
    }
}
```

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.

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.

The screenshot shows a web browser window with the URL `127.0.0.1:8000/datatest`. The page title is "Users Header Component". Below it, a heading says "This is the list of Users". A table lists five users:

ID	Name	Email	Operation	Operation
1	Sam Wrangler	samwrangler@gmail.com	Delete	Update
2	Peter Wriler	peterwriler@outlook.com	Delete	Update
3	Ronan Nalley	ronannalley@yahoo.com	Delete	Update
4	Yvonne Monahen	yvonnemonahen@gmail.com	Delete	Update
5	Brian Keller	briankeller@outlook.com	Delete	Update

Below the table are navigation links: "« Previous" and "Next »". The text "Showing 1 to 5 of 15 results" is displayed, followed by page numbers "1 2 3".

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.

The screenshot shows a code editor with the file `updateUser.blade.php` open. The left sidebar shows a project structure with "PRACTICAL2" selected, containing "views", "routes", and other files like `header.blade.php`, `about.blade.php`, etc. The main editor area contains the following PHP code:

```

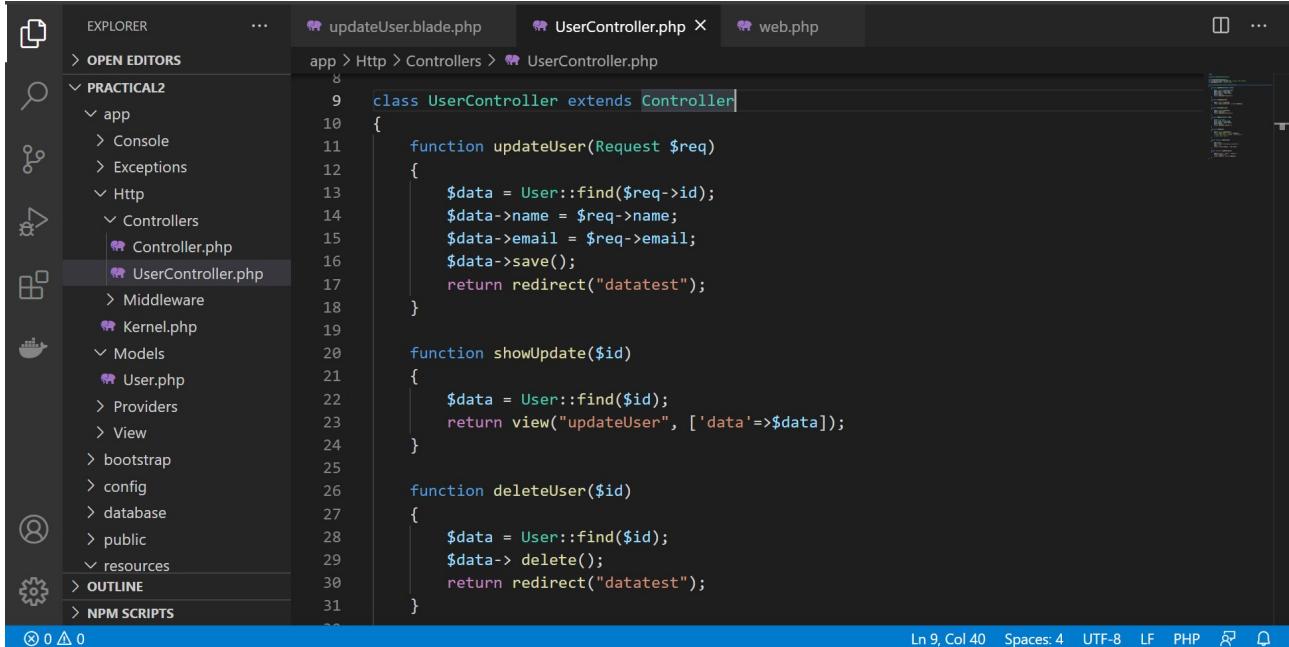
<x-header data="Update User" />
<form action="updateUser" method="POST">
    @csrf
    <input type="hidden" name="id" value="{{ $data['id'] }}>
    <input type="text" name="name" value="{{ $data['name'] }}> <br> <br>
    <input type="text" name="email" value="{{ $data['email'] }}> <br> <br>
    <button type = "submit"> Update User </button>
</form>

```

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.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

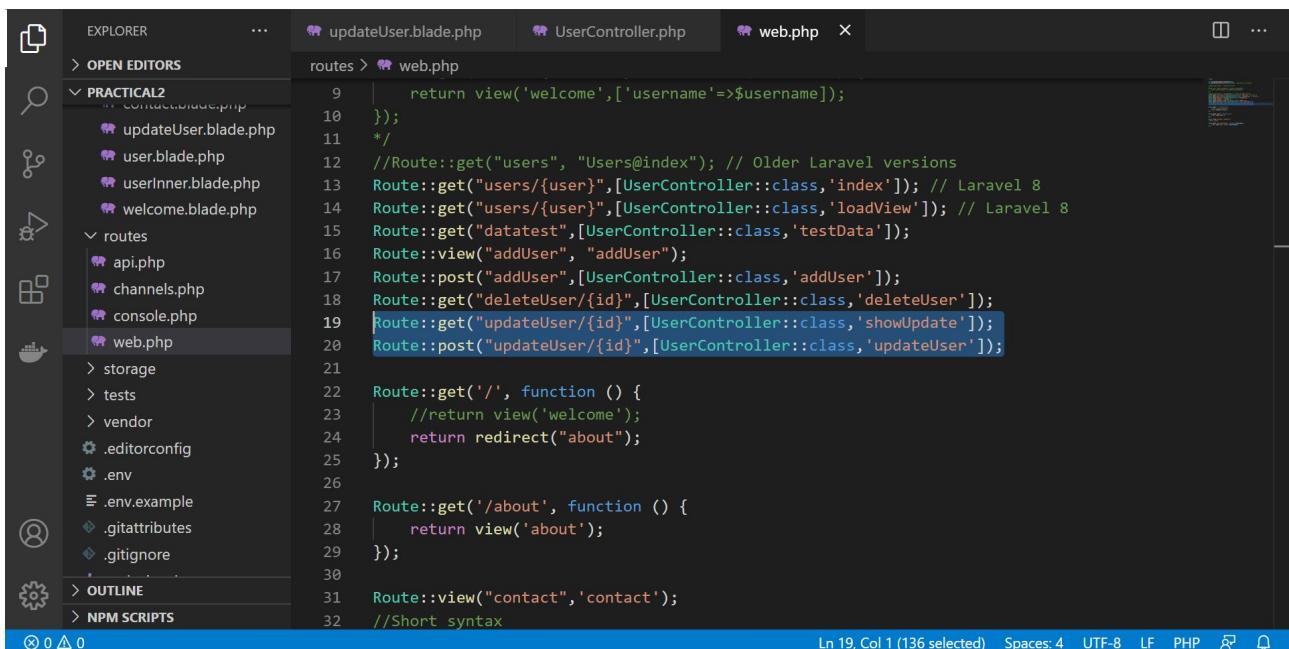


The screenshot shows the VS Code interface with the following details:

- Explorer View:** Shows the project structure under the 'PRACTICAL2' folder, including 'app', 'Http', 'Models', 'Providers', 'View', 'bootstrap', 'config', 'database', 'public', and 'resources'.
- Editor View:** Displays the contents of the 'UserController.php' file. The code defines a class 'UserController' extending 'Controller'. It contains three methods: 'updateUser', 'showUpdate', and 'deleteUser'. The 'updateUser' method finds a user by ID, updates their name and email, saves changes, and redirects to 'datatest'. The 'showUpdate' method finds a user by ID and returns a view named 'updateUser' with the user data. The 'deleteUser' method finds a user by ID and deletes it, then redirects to 'datatest'.
- Bottom Status Bar:** Shows 'Ln 9, Col 40' and other file-related information.

Figure 16: updateUser logics in UserController.

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



The screenshot shows the VS Code interface with the following details:

- Explorer View:** Shows the project structure under the 'PRACTICAL2' folder, including 'CONTROLLER.blade.php', 'updateUser.blade.php', 'user.blade.php', 'userInner.blade.php', 'welcome.blade.php', 'routes', 'api.php', 'channels.php', 'console.php', and 'web.php'.
- Editor View:** Displays the contents of the 'routes/web.php' file. It contains route definitions for various URLs, including 'welcome', 'users', 'addUser', 'deleteUser', 'updateUser', and 'about'. The 'updateUser' route is highlighted with a blue selection bar.
- Bottom Status Bar:** Shows 'Ln 19, Col 1 (136 selected)' and other file-related information.

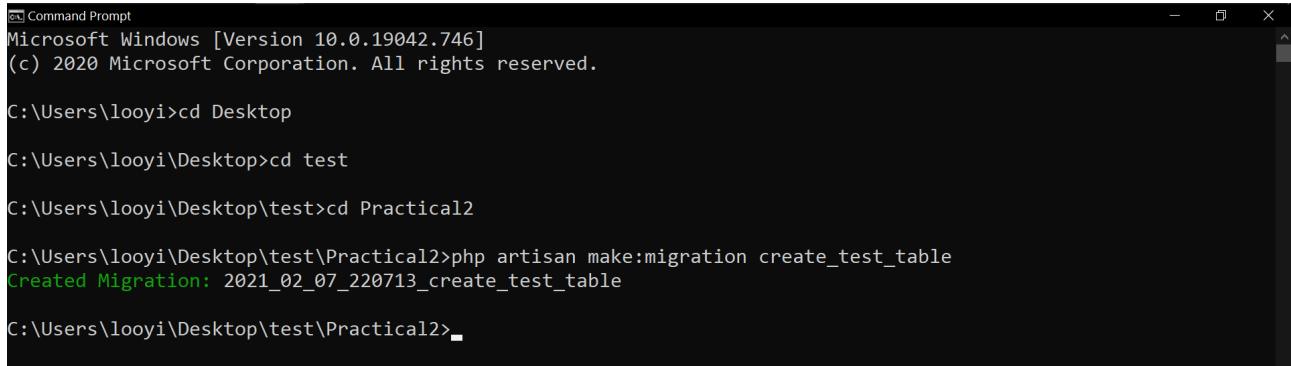
Figure 17: updateUser routes to UserController.

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

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.

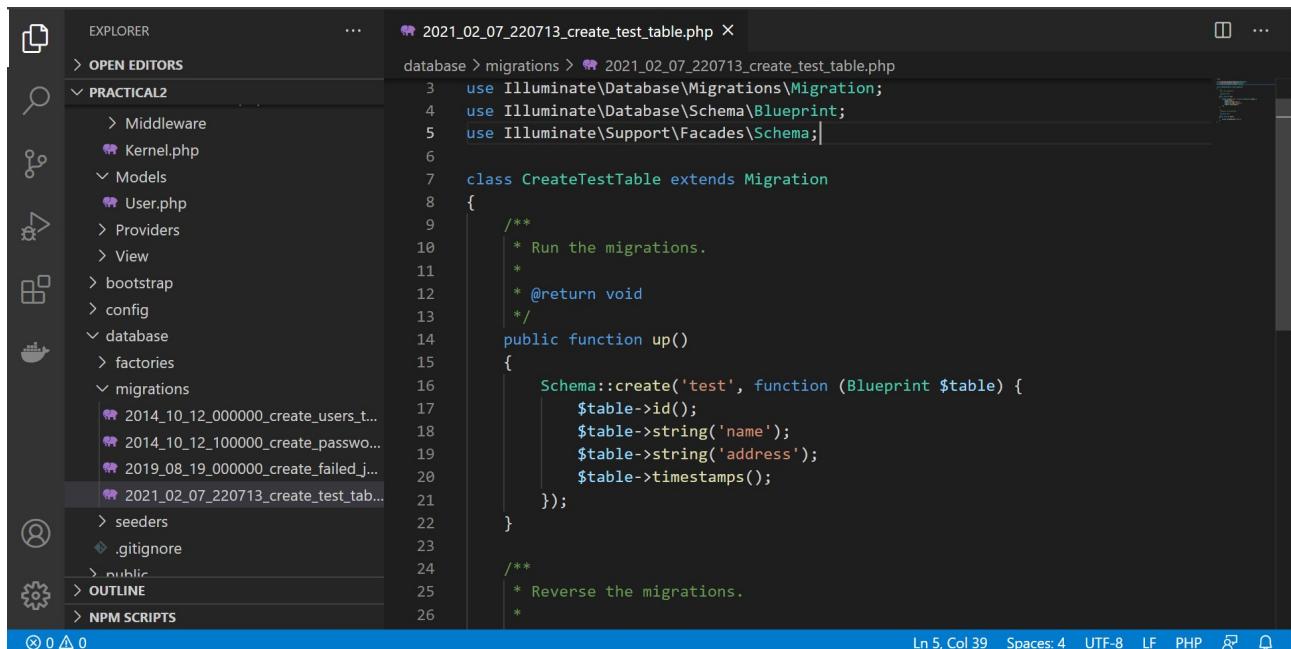


```
Command Prompt
Microsoft Windows [Version 10.0.19042.746]
(c) 2020 Microsoft Corporation. All rights reserved.

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.

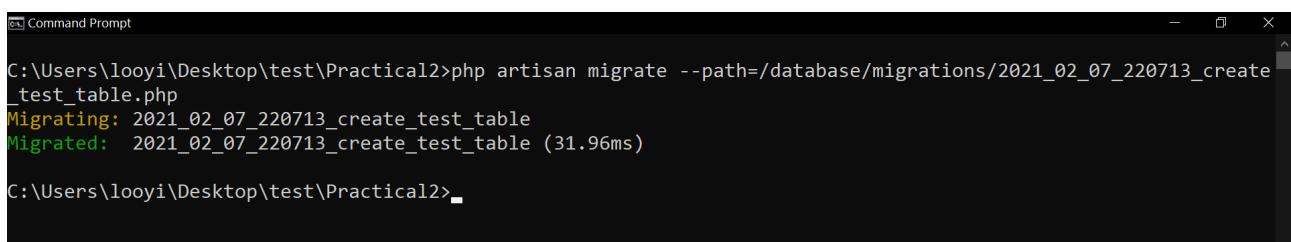


```
EXPLORER          2021_02_07_220713_create_test_table.php X
...                ...
OPEN EDITORS
PRACTICAL2
  > Middleware
  < Kernel.php
  > Models
    < User.php
  > Providers
  > View
  > bootstrap
  > config
  > database
    > factories
    > migrations
      < 2014_10_12_000000_create_users...
      < 2014_10_12_100000_create_passwo...
      < 2019_08_19_000000_create_failed_j...
      < 2021_02_07_220713_create_test_ta...
    > seeders
    < .gitignore
  > public
  > OUTLINE
  > NPM SCRIPTS
  < 0 Δ 0
```

```
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     *
```

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.



```
Command Prompt
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>
```

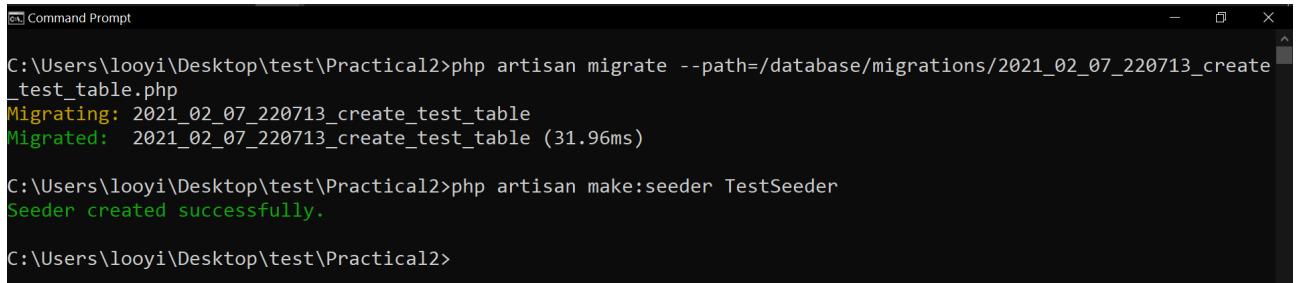
Figure 20: Migrate specific table with Artisan CLI.

UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT

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.



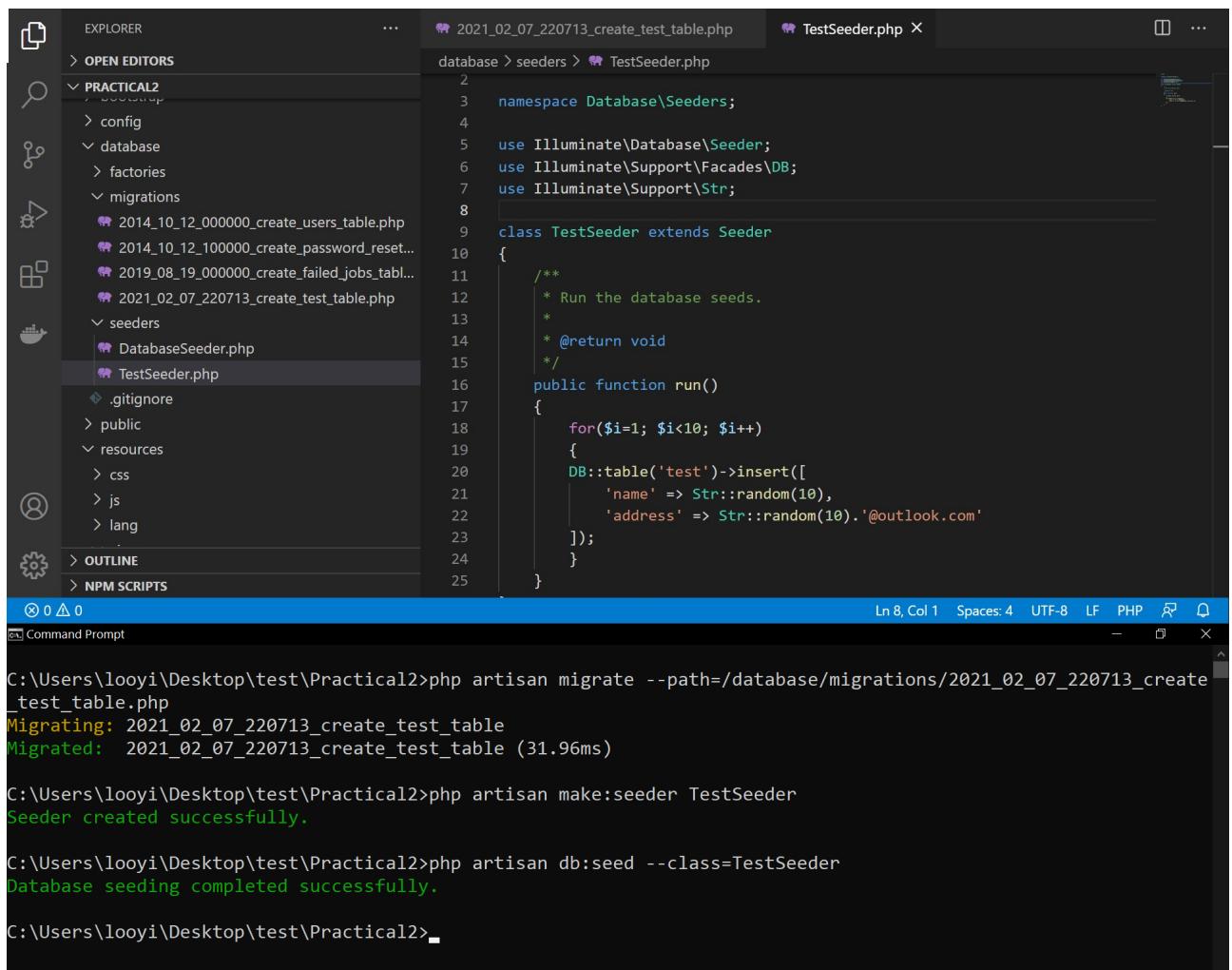
```
C:\Users\looyi\Desktop\test\Practical2>php artisan migrate --path=/database/migrations/2021_02_07_220713_create_test_table.php
Migrating: 2021_02_07_220713_create_test_table
Migrated: 2021_02_07_220713_create_test_table (31.96ms)

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.



The screenshot shows a code editor with a sidebar and a command prompt at the bottom.

Code Editor:

- Explorer sidebar shows a project structure with folders like config, database, factories, migrations, and seeders.
- The seeders folder contains two files: DatabaseSeeder.php and TestSeeder.php.
- The TestSeeder.php file is open in the editor, showing the following code:

```
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      *
14      * @return void
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     }
}
```

Command Prompt:

```
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.