CERTIFICATE

This is to certify that Mr./Ms. ***…..... Hemil…Chovatiya.....*** *with enrolment no.* ***.....200303108003.....*** *has successfully completed* ***his****/her laboratory experiments in the* ***Web Development&Framework Lab*** *from the department of* ***.....Information Technology(6ITA1)…….....*** *during the academic year* ***.....2022-2023.....***

Date of Submission: ..... Staff In charge: .....

Head of Department: .....

INDEX

Sr. No	Experiment Title	Page No		Date of Performance	Date of Assessment	Marks (out of 10)	Sign
		From	To				
1	Write a code to demonstrate Laravel Fundamentals						
2	Demonstrate the practical of prepping the database a. Implement Database migration b. Implement eloquent models c. Implement eloquent relationships						
3	Build a REST API with Laravel						
4	Implement routing in Laravel a. Displaying a view b. Authentication c. The Task controller						
5	Demonstrate how to build layouts and Views a. Defining the layout b. Defining the child view						
6	Implement adding tasks, displaying existing tasks and deleting tasks.						
7	Process Payments with Stripe and Laravel Cashier						
8	Authentication with Laravel						
9	Demonstrate How It's Made: Laravel Router						
10	Build a CMS with Laravel						

PRACTICAL-1

AIM: Write a code to demonstrate Laravel Fundamentals.

Code:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
   public function index()
   {
      $users = \App\Models\User::all();
      return view('users.index', ['users' => $users]);
   }

   public function create()
   {
      return view('users.create');
   }

   public function store(Request $request)
   {
      $user = new \App\Models\User;
      $user->name = $request->name;
      $user->email = $request->email;
      $user->password = Hash::make($request->password);
      $user->save();

      return redirect('/users');
   }

   public function show($id)
   {
      $user = \App\Models\User::find($id);
      return view('users.show', ['user' => $user]);
   }

   public function edit($id)
   {
      $user = \App\Models\User::find($id);
      return view('users.edit', ['user' => $user]);
   }
```

```
    public function update(Request $request, $id)
    {
        $user = \App\Models\User::find($id);
        $user->name = $request->name;
        $user->email = $request->email;
        if ($request->password) {
            $user->password = Hash::make($request->password);
        }
        $user->save();

        return redirect('/users/' . $id);
    }

    public function destroy($id)
    {
        $user = \App\Models\User::find($id);
        $user->delete();

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

This code defines a UserController that handles HTTP requests related to users. It includes methods for displaying a list of all users, creating a new user, storing a new user in the database, showing the details of a specific user, editing a specific user, updating a specific user's information, and deleting a specific user.

The index method retrieves all users from the database using the all method of the User model and passes them to a view called users.index. The create method simply returns a view called users.create. The store method creates a new User instance, sets its attributes based on the values submitted in a POST request, saves the new user to the database, and then redirects the user to the index page for all users.

The show method retrieves a specific user from the database using the find method of the User model and passes it to a view called users.show. The edit method is similar, but it returns a view called users.edit instead.

The update method retrieves a specific user from the database, updates its attributes based on the values submitted in a POST request, saves the updated user to the database, and then redirects the user to the show page for that user.

The destroy method retrieves a specific user from the database and deletes it, then redirects the user to the index page for all users.

PRACTICAL-2

AIM: Demonstrate the practical of prepping the database

a. Implement Database migration
b. Implement eloquent models
c. Implement eloquent relationships.

Code:

a. Implementing database migration

First, we'll create a migration to create a users table in our database. We'll use the following command in our terminal:

php artisan make:migration create_users_table --create=users

This will create a new migration file in the database/migrations directory. We can then edit the up method of this file to define the structure of our users table:

```
public function up()
{
   Schema::create('users', function (Blueprint $table) {
      $table->id();
      $table->string('name');
      $table->string('email')->unique();
      $table->timestamp('email_verified_at')->nullable();
      $table->string('password');
      $table->rememberToken();
      $table->timestamps();
   });
}
```

This will create a table with columns for the user's name, email, email verification timestamp, password, and remember token, as well as timestamps for when the user was created and updated.

To run this migration and create the users table in our database, we can use the following command in our terminal:

php artisan migrate

b. Implementing eloquent models

Now that we have our users table in our database, we can create an Eloquent model to interact with it. We'll use the following command in our terminal:

php artisan make:model User

This will create a new User model in the app/Models directory. We can then define the properties and methods of this model to interact with the users table:

```
namespace App\Models;

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

class User extends Model
{
    use HasFactory;

    protected $fillable = [
        'name',
        'email',
        'password',
    ];
}
```

Here, we're using the HasFactory trait to give us some useful Eloquent methods, and we're specifying the fillable property to allow mass assignment of the name, email, and password attributes.

c. Implementing eloquent relationships

Finally, let's implement some Eloquent relationships between our User model and other models. For example, let's say we want to create a posts table and a Post model to represent blog posts. We can create a migration for the posts table using the following command in our terminal:

php artisan make:migration create_posts_table --create=posts

We can then define the structure of this table in the up method of the migration, and run the migration using php artisan migrate.

Next, we can create a Post model using the php artisan make:model Post command, and define its properties and methods to interact with the posts table.

To create a one-to-many relationship between User and Post, we can add the following method to the User model:

```
public function posts()
{
   return $this->hasMany(Post::class);
}
```

This tells Laravel that a User can have many Post instances, and that we can access them using the posts method on a User instance.

To create a many-to-one relationship from Post to User, we can add the following method to the Post model:

```
public function user()
{
   return $this->belongsTo(User::class);
}
```

This tells Laravel that a Post belongs to a User instance, and that we can access the User instance

PRACTICAL-3

AIM: Build a REST API with Laravel.

Code:

1. Set up a new Laravel project

First, we'll create a new Laravel project by running the following command in our terminal:

composer create-project --prefer-dist laravel/laravel api-example

This will create a new Laravel project in a directory called api-example. We can then navigate to this directory and run the following command to start the Laravel development server:

php artisan serve

2. Create a database and migrate the tables

Next, we'll create a new MySQL database and update the .env file in our Laravel project with our database credentials.

Once our database is set up, we can create a new migration for our products table by running the following command in our terminal:

php artisan make:migration create_products_table --create=products

We can then define the structure of this table in the up method of the migration. For example, we might use the following code to define a products table with id, name, description, and price columns:

```
public function up()
{
   Schema::create('products', function (Blueprint $table) {
      $table->id();
      $table->string('name');
      $table->text('description')->nullable();
      $table->decimal('price', 8, 2);
      $table->timestamps();
   });
}
```

Once our migration is defined, we can run it by running the following command in our terminal:

php artisan migrate

This will create the products table in our database.

3. Define a product model

Next, we'll create a new Eloquent model for our products table. We can create this model by running the following command in our terminal:

php artisan make:model Product

This will create a new Product model in the app directory. We can define the properties and methods of this model to interact with the products table.

For example, we might use the following code to define a Product model with name, description, and price attributes:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
   protected $fillable = ['name', 'description', 'price'];
}
```

4.Define routes and controllers

Now that our database and model are set up, we can define routes and controllers to handle HTTP requests and responses.

First, we'll define a route for getting all products. We can do this by adding the following code to our routes/api.php file:

```
use App\Product;

Route::get('/products', function () {
   return Product::all();
});
```

This will define a GET route for /products that returns all products from our database.

Next, we'll define a route for getting a specific product by ID. We can do this by adding the following code to our routes/api.php file:

```
use App\Product;

Route::get('/products/{id}', function ($id) {
    return Product::find($id);
});
```

This will define a GET route for /products/{id} that returns the product with the specified ID.

Finally, we'll define a route for creating a new product. We can do this by adding the following code to our routes/api.php file:

```
use App\Product;
use Illuminate\Http\Request;

Route::post('/products', function (Request $request) {
    return Product::create($request->all());
});
```

This will define a POST route for /products that creates a new product using the data

PRACTICAL-4

AIM: . Implement routing in Laravel

a. Displaying a view
b. Authentication
c. The Task controller .

Code:

a. Displaying a view:

Define a route in your web.php file in the routes directory. For example:

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

In the above code, the first parameter is the URL path and the second parameter is a closure function that returns a view. In this case, the view being returned is "welcome.blade.php".

Create a new file called "welcome.blade.php" in the "resources/views" directory. This file will contain the HTML code for your view.

b. Authentication:

Laravel provides built-in authentication features that you can use by running the following command:

```
php artisan make:auth
```

This command will create the necessary views and controllers for authentication, as well as the necessary routes.

You can customize the authentication views by modifying the files in the "resources/views/auth" directory.

To protect certain routes so that only authenticated users can access them, use the "middleware" function in your route definition. For example:

```
Route::get('/dashboard', function () {
   // Only authenticated users can access this route
})->middleware('auth');
```

c. The Task controller:

1.Create a new controller by running the following command:

php artisan make:controller TaskController

2.This will create a new file called "TaskController.php" in the "app/Http/Controllers" directory.

3.Define your controller methods in the TaskController file. For example:

```
public function index()
{
   // Return a list of tasks
}

public function create()
{
   // Display a form for creating a new task
}

public function store(Request $request)
{
   // Store the new task in the database
}

public function show($id)
{
   // Display the details of a specific task
}

public function edit($id)
{
   // Display a form for editing a task
}

public function update(Request $request, $id)
{
   // Update the task in the database
}

public function destroy($id)
{
   // Delete the task from the database
```

}

4.Define routes for your controller methods in the web.php file. For example:

```
Route::get('/tasks', [TaskController::class, 'index']);
Route::get('/tasks/create', [TaskController::class, 'create']);
Route::post('/tasks', [TaskController::class, 'store']);
Route::get('/tasks/{id}', [TaskController::class, 'show']);
Route::get('/tasks/{id}/edit', [TaskController::class, 'edit']);
Route::put('/tasks/{id}', [TaskController::class, 'update']);
Route::delete('/tasks/{id}', [TaskController::class, 'destroy']);
```

5.In the above code, the first parameter is the URL path and the second parameter is an array that specifies the controller method to be called for that route. For example, "[TaskController::class, 'index']" specifies that the "index" method of the "TaskController" class should be called when the "/tasks" route is accessed.

PRACTICAL-5

AIM: . Demonstrate how to build layouts and Views

a. Defining the layout

b. Defining the child view .

Code:

a. Defining the layout:

Create a new XML file in your app's "res/layout" directory, e.g. "my_layout.xml".
Open the file and add the following code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

</LinearLayout>
```

This creates a LinearLayout, which is a common layout container in Android that arranges child views vertically or horizontally.

b.Defining the child view:

Inside the LinearLayout tags, add a TextView element with some text, e.g.:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello, World!" />

</LinearLayout>
```

This creates a TextView child view with the text "Hello, World!". You can customize its appearance using various attributes such as "android:textSize" and "android:textColor".

That's it! You can now use this layout in your app's activities or fragments by inflating it with LayoutInflater, e.g.:

```
class MyActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.my_layout)
    }
}
```

This will set the activity's content view to the LinearLayout with the TextView child view.

PRACTICAL-6

AIM: Implement adding tasks, displaying existing tasks and deleting tasks.

Code:

First, let's create a new Laravel project and set up a database connection:

```
composer create-project --prefer-dist laravel/laravel task-manager
cd task-manager
cp .env.example .env
php artisan key:generate
```

Update the DB_* variables in .env to match your database settings.

Next, let's create a Task model and migration:

```
php artisan make:model Task -m
```

This will create a Task model and a migration file for the tasks table.

Update the up method in the migration file to create the necessary columns:

```
public function up()
{
   Schema::create('tasks', function (Blueprint $table) {
      $table->id();
      $table->string('title');
      $table->text('description');
      $table->boolean('completed')->default(false);
      $table->timestamps();
   });
}
```

Now, run the migration to create the tasks table:

```
php artisan migrate
```

Next, let's create a TaskController to handle the CRUD operations:

```
php artisan make:controller TaskController --resource
```

This will create a TaskController with resourceful methods for handling CRUD operations.

Add the following code to the TaskController:

```
use App\Models\Task;
use Illuminate\Http\Request;

public function index()
{
   $tasks = Task::orderBy('created_at', 'desc')->get();
   return view('tasks.index', compact('tasks'));
}

public function create()
{
   return view('tasks.create');
}

public function store(Request $request)
{
   $validatedData = $request->validate([
      'title' => 'required|max:255',
      'description' => 'required',
   ]);

   $task = new Task();
   $task->title = $validatedData['title'];
   $task->description = $validatedData['description'];
   $task->save();

   return redirect()->route('tasks.index');
}

public function destroy(Task $task)
{
   $task->delete();
   return redirect()->route('tasks.index');
}
```

The index method fetches all the tasks from the database and returns them to the tasks.index view. The create method returns a view with a form for creating a new task. The store method validates the form data and creates a new task in the database. The destroy method deletes a task from the database.

Next, let's create the views for displaying and creating tasks.

Create a new file resources/views/tasks/index.blade.php with the following content:

```
@extends('layouts.app')

@section('content')
  <div class="row">
    <div class="col-lg-12 margin-tb">
      <div class="pull-left">
        <h2>Task Manager</h2>
      </div>
      <div class="pull-right">
        <a class="btn btn-success" href="{{ route('tasks.create') }}"> Create New Task</a>
      </div>
    </div>
  </div>

  @if (session('status'))
    <div class="alert alert-success">
      {{ session('status') }}
    </div>
  @endif

  <table class="table table-bordered">
    <tr>
      <th>Title</th>
      <th>Description</th>
      <th>Actions</th>
    </tr>
    @foreach ($tasks as $task)
    <tr>
      <td>{{ $task->title }}</td>
      <td>{{ $task->description }}</td>
```

PRACTICAL-7

AIM: Process Payments with Stripe and Laravel Cashier .

Code:

Step 1: Set up a Stripe account
If you don't have a Stripe account already, head to stripe.com and create one. Once you've created your account, navigate to the Stripe Dashboard and go to the "Developers" tab to obtain your API keys.

Step 2: Install Laravel Cashier
Next, you'll need to install Laravel Cashier using Composer. Open up your terminal and run the following command:

composer require laravel/cashier

This will install the latest version of Laravel Cashier into your project.

Step 3: Configure Laravel Cashier
Next, you'll need to configure Laravel Cashier with your Stripe API keys. Open up your .env file and add the following lines:

STRIPE_KEY=your_stripe_publishable_key
STRIPE_SECRET=your_stripe_secret_key

Replace "your_stripe_publishable_key" and "your_stripe_secret_key" with the actual values from your Stripe Dashboard.

Step 4: Set up a User model
To use Laravel Cashier, you'll need to have a User model with a Stripe customer ID column. If you don't already have a User model, create one using the following command:

php artisan make:model User

Then, add a stripe_id column to your users table by creating a migration:

php artisan make:migration add_stripe_id_to_users_table --table=users

In the migration file, add the following line to add the stripe_id column:

$table->string('stripe_id')->nullable();

Then, run the migration to apply the changes to your database:

php artisan migrate

Step 5: Create a Stripe Customer
When a user signs up on your website, you'll need to create a Stripe customer for them. You can do this using Laravel Cashier's createAsStripeCustomer method. Here's an example of how to use it:

```
$user = User::find(1);
$user->createAsStripeCustomer();
```

This will create a Stripe customer with the user's email address and associate it with their user account.

Step 6: Create a Payment Form
To collect payment from your users, you'll need to create a payment form on your website. Here's an example of a simple payment form:

```
<form action="/charge" method="POST">
  @csrf
  <div class="form-group">
    <label for="amount">Amount</label>
    <input type="number" id="amount" name="amount" class="form-control" required>
  </div>
  <div class="form-group">
    <label for="card-element">Credit or debit card</label>
    <div id="card-element">
      <!-- A Stripe Element will be inserted here. -->
    </div>

    <!-- Used to display Element errors. -->
    <div id="card-errors" role="alert"></div>
  </div>
  <button type="submit" class="btn btn-primary">Submit Payment</button>
</form>
```

This form collects the payment amount and the user's credit card details. Note that you'll need to include the Stripe Elements JavaScript library to use this form.

Step 7: Process the Payment
When the user submits the payment form, you'll need to process the payment on the server side using Laravel Cashier's charge method. Here's an example of how to use it:

```
public function charge(Request $request)
{
    $user = Auth::user();

    $amount = $request->input('amount');

    // Process payment using the $amount value

    return view('payment_success');
}
```

PRACTICAL-8

AIM: Authentication with Laravel

Code:

Laravel is a popular PHP web framework that provides many built-in tools and features for developing web applications, including authentication. Laravel makes it easy to authenticate users with built-in authentication scaffolding and a robust set of authentication features.

To get started with authentication in Laravel, follow these steps:

Set up a new Laravel application or use an existing one.
Generate the authentication scaffolding using the make:auth Artisan command. This command generates the necessary views, routes, and controllers for user authentication.
Run database migrations to create the necessary tables for authentication using the migrate Artisan command.
Set up your authentication guards and providers in the config/auth.php configuration file. Laravel provides several options for authentication, including session-based authentication, token-based authentication, and socialite authentication.
Use the Auth facade to authenticate users in your controllers or routes. The Auth facade provides methods for logging in, logging out, and checking if a user is authenticated.
Here's an example of how to authenticate a user using Laravel's built-in authentication:

```
use Illuminate\Support\Facades\Auth;

public function login(Request $request)
{
   $credentials = $request->only('email', 'password');

   if (Auth::attempt($credentials)) {
      // Authentication was successful
      return redirect()->intended('dashboard');
   } else {
      // Authentication failed
      return back()->withErrors([
         'email' => 'The provided credentials do not match our records.',
      ]);
   }
}
```

In this example, we're using the attempt method on the Auth facade to attempt to authenticate the user with the provided credentials. If the authentication is successful, we redirect the user

to the dashboard. If the authentication fails, we redirect the user back to the login form with an error message.

That's the basics of authentication with Laravel. There's a lot more you can do with authentication, such as customizing the authentication views or implementing two-factor authentication, but these steps should give you a good starting point.

PRACTICAL-9

AIM: Demonstrate How It's Made: Laravel Router .

Code:

Laravel's router is responsible for handling incoming HTTP requests and determining which code should be executed to respond to the request. It does this by matching the requested URL with a route that has been defined in the application.

Here's a simplified step-by-step process of how Laravel's router works:

The router receives an incoming HTTP request.

It examines the URL of the request to determine which route should handle the request. This is done by comparing the URL against the routes that have been defined in the application.

If a matching route is found, the router executes the code associated with that route. This code could be a closure, a controller method, or any other callable function.

If no matching route is found, the router will return a 404 error response.

Here's an example of how you can define a route in Laravel:

```
Route::get('/hello', function () {
    return 'Hello, World!';
});
```

This code defines a route that responds to GET requests to the /hello URL path. When a request is made to this URL path, the anonymous function defined in the second parameter will be executed, and it will return the string "Hello, World!" as the response.

Laravel's router has many other features that allow you to define more complex routes, such as route parameters, optional parameters, route groups, middleware, and more. However, the basic principle remains the same: the router matches incoming HTTP requests with defined routes and executes the appropriate code to respond to the request.

PRACTICAL-10

AIM: Build a CMS with Laravel.

Code:

Building a CMS (Content Management System) with Laravel involves creating a database-driven web application that allows users to create, manage, and publish content on their website. Here are the basic steps to build a CMS with Laravel:

Set up a new Laravel application or use an existing one.
Define the database schema for your CMS using migrations. For example, you may need tables for users, pages, posts, categories, tags, and comments.
Create models, controllers, and views for each type of content you want to manage. For example, you might have a Page model with a PagesController and a set of views for creating, editing, and displaying pages.
Implement authentication and authorization to restrict access to the CMS to only authorized users. Laravel provides built-in authentication scaffolding that you can customize to fit your needs.
Implement CRUD (Create, Read, Update, Delete) operations for each type of content you want to manage. This involves defining routes, controller methods, and views for creating, editing, and deleting content, as well as displaying lists of content.
Implement search and filtering functionality to allow users to find specific content based on criteria such as keywords, categories, or dates.
Implement pagination to display long lists of content in smaller, more manageable chunks.
Implement validation to ensure that content is entered in the correct format and meets any requirements you have set.
Implement caching and optimization techniques to improve performance and reduce server load.
Test your CMS thoroughly to ensure it works as expected and is secure.
Here's an example of how to create a simple CMS with Laravel that allows users to manage pages:

1.Define the database schema using migrations:

php artisan make:migration create_pages_table --create=pages

```
Schema::create('pages', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('slug')->unique();
    $table->text('content');
    $table->timestamps();
});
```

2.Create a Page model:

php artisan make:model Page

3.Create a PagesController with CRUD operations:

php artisan make:controller PagesController --resource

4. Define routes for managing pages:

Route::resource('pages', PagesController::class);

5. Create views for managing pages, including a form for creating/editing pages and a view for displaying a list of pages.

6. Implement authentication and authorization to restrict access to the CMS to only authorized users. Laravel provides built-in authentication scaffolding that you can customize to fit your needs.

7. Implement search and filtering functionality to allow users to find specific pages based on criteria such as keywords or categories.

8. Implement pagination to display long lists of pages in smaller, more manageable chunks.

9. Implement validation to ensure that pages are entered in the correct format and meet any requirements you have set.

10. Test your CMS thoroughly to ensure it works as expected and is secure.

11. This is just a basic example of how to build a CMS with Laravel. You can customize it to fit your specific needs by adding more types of content, features, and functionality.