

Form Create

Form Validation

Form data fetch with post method using @csrf

How to route one page to another

@csrf Method in Laravel :-

CSRF refers to Cross Site Forgery attacks on web applications. CSRF attacks are the unauthorized activities which the authenticated users of the system perform. As such, many web applications are prone to these attacks.

Laravel offers CSRF protection in the following way –

Laravel includes an in built CSRF plug-in, that generates tokens for each active user session. These tokens verify that the operations or requests are sent by the concerned authenticated user.

```
<form method = "POST" action="/profile">  
  @csrf  
  ...  
</form>
```

Now Follow the steps to using Csrf Method:-

Step 1:- create a controller:-

```
PS C:\xampp\htdocs\ankita> php artisan make:Controller Users  
  
INFO Controller [C:\xampp\htdocs\ankita\app\Http\Controllers/Users.php] created successfully.
```

```

app > Http > Controllers > Users.php > ...
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class Users extends Controller
8  {
9      public function dataget(Request $req){
10         return $req->input();
11     }
12
13 }
14

```

Step 2:- Create a form using the POST method in view folder using form.blade.php and using the @csrf Hidden token

```

resources > views > form.blade.php > ...
1  <!DOCTYPE html>
2  <html>
3  <body>
4
5  <h2>HTML Forms</h2>
6
7  <form action="user" method="POST">
8      @csrf
9      <label for="fname">First name:</label><br>
10     <input type="text" id="fname" name="fname" value="John"><br>
11     <label for="lname">Last name:</label><br>
12     <input type="text" id="lname" name="lname" value="Doe"><br><br>
13     <input type="submit" value="Submit">
14 </form>
15 </body>
16 </html>
17
18

```

Step 3:- create routes for access the form file :-

```
Users.php  web.php  .env  f
routes > web.php
17 |         return view('welcome');
18 |     });
19 |
20 |     Route::get('/home', function () {
21 |         return view('welcome');
22 |     });
23 |
24 |
25 |     Route::View('login', 'form');
26 |
```

Step 4:- now check the form in browser type login and then see:-



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/login'. The page title is 'HTML Forms'. Below the title, there is a form with two text input fields. The first field is labeled 'First name:' and contains the text 'Ankita'. The second field is labeled 'Last name:' and contains the text 'Shukla'. Below these fields is a 'Submit' button.

Step 5:- Create a Route for accessing the enter data in form

Add the controller namespace in web.php

```
Users.php web.php .env form.blade.php
routes > web.php
1 <?php
2
3 use Illuminate\Support\Facades\Route;
4 use App\Http\Controllers\Users;
5 /*
6 |-----
7 | Web Routes
8 |-----
9 |
10 | Here is where you can register web routes for your application. These
11 | routes are loaded by the RouteServiceProvider and all of them will
12 | be assigned to the "web" middleware group. Make something great!
13 |
14 */
15
```

Create a route with post and use url path and define controller class

```
26
27
28 Route::post('user',[Users::class,'dataget']);
29
30
31
```

And final you see the output

```
< 127.0.0.1:8000/user
{"_token":"FPIdkeUEkpxfszF6ymBna0SqOD0qQYgMcZQWYj6a","fname":"Ankita","lname":"Shukla"}
```

Using Validation in form

```
Users.php X web.php .env form.blade.php
app > Http > Controllers > Users.php > Users
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class Users extends Controller
8  {
9      public function dataget(Request $req){
10
11          $req->validate(
12              [
13                  'fname'=>'required',
14                  'lname'=>'required'];
15          return $req->all();
16      }
17
18  }
```

```
resources > views > form.blade.php > html > body
1  <!DOCTYPE html>
2  <html>
3  <body>
4
5  <h2>HTML Forms</h2>
6
7  <form action="user" method="POST">
8      @csrf
9      <label for="fname">First name:</label><br>
10     <input type="text" id="fname" name="fname" value="{{old('fname')}}"><br>
11     <span class="text-danger">
12         @error('fname')
13             {{$message}}
14         @enderror
15     </span>
16     <br>
17
18     <label for="lname">Last name:</label><br>
19     <input type="text" id="lname" name="lname" value="{{old('lname')}}"><br><br>
20     <span class="text-danger">
21         @error('lname')
22             {{$message}}
23         @enderror
24     </span>
25     <input type="submit" value="Submit">
26 </form>
27 </body>
28 </html>
```

How to route one page to another :-

```
<div class="collapse navbar-collapse" id="navbarSupportedContent">
  <ul class="navbar-nav me-auto mb-2 mb-lg-0">
    <li class="nav-item">
      <a class="nav-link active" aria-current="page" href="{{ route('home') }}">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="{{ route('about') }}">About</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="{{ route('contact') }}">Contact</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="{{ route('services') }}">Services</a>
    </li>
  </ul>
  <form class="d-flex" role="search">
    <input class="form-control me-2" type="search" placeholder="Search" aria-label="Search">
    <button class="btn btn-outline-success" type="submit">Search</button>
  </form>
</div>
```

Set route path :-

```
Route::get('/nature', [BladeController::class, 'home'])->name('home');
Route::get('/about', [BladeController::class, 'about'])->name('about');
Route::get('/services', [BladeController::class, 'services'])->name('services');
Route::get('/contact', [BladeController::class, 'contact'])->name('contact');
```

What is .env file/set .env file

What is configuration file

Caching of configuration file

Database connectivity

What is migration

Commands of migration

Create migration

Maintenance Mode

What is .env File

In Laravel, the .env file is a configuration file used to store sensitive information and environmental variables for your application. It stands for "environment" file. This file is located in the root directory of your Laravel project.

The .env file uses a key-value pair format, where each line represents a configuration variable. It typically contains sensitive information such as database credentials, API keys, and other environment-specific settings. These values can be accessed throughout your Laravel application using the env helper function or by using the `$_ENV` superglobal.

Here's an example of what a .env file in Laravel might look like:

```
APP_NAME=Laravel
APP_ENV=production
APP_KEY=your-application-key
APP_DEBUG=false

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=your-database-name
DB_USERNAME=your-database-username
DB_PASSWORD=your-database-password
```

What are configuration files in Laravel

In Laravel, configuration files are files that contain various settings and options for your application. These files allow you to customize and define the behavior of different components and services within Laravel.

Configuration files in Laravel are typically stored in the config directory of your Laravel project. They are written in PHP and follow a simple array-based structure.

Here are some examples of configuration files commonly found in Laravel:

1) app.php: This file contains general application settings such as the application name, environment, timezone, and service providers.

2) database.php: This file contains database connection configurations, including the default database connection, MySQL, PostgreSQL, SQLite, and more.

3) cache.php: This file contains configuration options for caching, such as the default cache driver, cache stores, and cache lifetimes.

4) mail.php: This file contains configurations related to sending emails, including the default mail driver, SMTP settings, and email encryption.

5) filesystems.php: This file contains configuration options for file storage and handling, including the default file storage driver, local disks, cloud storage providers, and their respective settings.

6) logging.php: This file contains configuration options for logging, including the log channels, their respective drivers, log levels, and log storage locations.

7) session.php: This file contains configuration options for session management, including the session driver, session lifetime, and session encryption.

These are just a few examples, and Laravel provides many more configuration files for different components and services. You can also create your own custom configuration files to define settings specific to your application.

To access the values defined in configuration files within your application, you can use the config helper function or the Config facade. For example, `config('app.name')` will retrieve the value of the name key from the app.php configuration file.

By utilizing configuration files, Laravel allows you to easily manage and customize various aspects of your application's behavior without modifying the underlying code, promoting flexibility and maintainability.

Caching of Configuration

In Laravel, you have the option to cache your application's configuration files, which can improve the performance of your application by reducing the overhead of reading and parsing configuration files on each request.

When you cache the configuration, Laravel will merge all the configuration files into a single file, which is then stored in the bootstrap/cache directory. This cached configuration file is loaded much faster than parsing multiple configuration files, resulting in quicker application startup times.

To cache your configuration, you can use the `config:cache` Artisan command. Open your command-line interface, navigate to the root directory of your Laravel project, and run the following command:

```
php artisan config:cache
```

This command will generate the cached configuration file `bootstrap/cache/config.php`. You should run this command whenever you make changes to your configuration files or deploy your application to ensure that the cached configuration is up to date.

It's important to note that if you make changes to your configuration files after caching, you'll need to clear the configuration cache by running the `config:clear` Artisan command:

```
php artisan config:clear
```

This command will remove the cached configuration file, and your application will revert to reading and parsing the individual configuration files on each request.

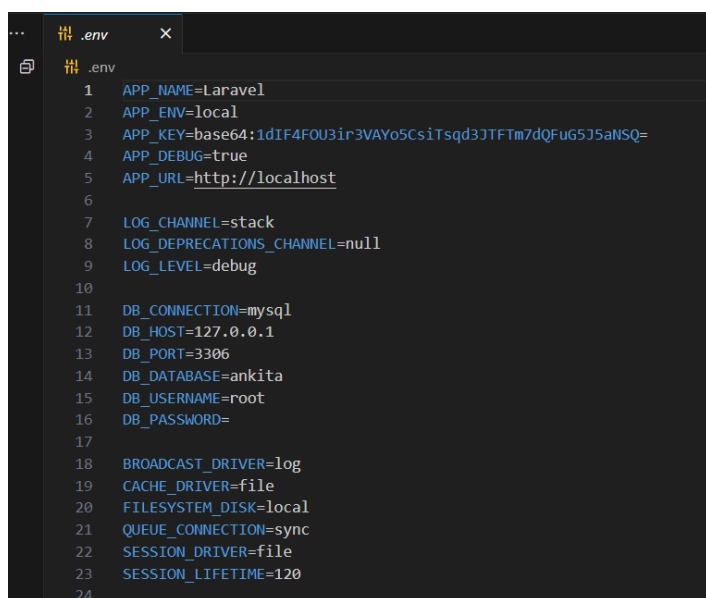
By caching your configuration, you can optimize the performance of your Laravel application by reducing the overhead of loading and parsing configuration files, especially in production environments where the configuration is less likely to change frequently.

Learn how to configure database connections in Laravel:-

Open the .env file in the root directory of your Laravel project. This file contains environment-specific variables, including database connection details.

Locate the following variables in the .env file:

- 1) DB_CONNECTION: This variable specifies the database connection driver. The default driver is usually set to mysql.
- 2) DB_HOST: This variable specifies the database host.
- 3) DB_PORT: This variable specifies the port number on which the database server is running.
- 4) DB_DATABASE: This variable specifies the name of the database you want to connect to.
- 5) DB_USERNAME: This variable specifies the username for the database connection.
- 6) DB_PASSWORD: This variable specifies the password for the database connection.
- 7) Update the values of these variables according to your database configuration. For example:

A screenshot of a code editor window showing the .env file. The file contains various environment variables for a Laravel application. The database configuration variables are highlighted in the image.

```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:1dIF4FOU3ir3VAYo5CsiTsqd3JTFTm7dQFug5J5aNSQ=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=ankita
15 DB_USERNAME=root
16 DB_PASSWORD=
17
18 BROADCAST_DRIVER=log
19 CACHE_DRIVER=file
20 FILESYSTEM_DISK=local
21 QUEUE_CONNECTION=sync
22 SESSION_DRIVER=file
23 SESSION_LIFETIME=120
24
```

Save the changes to the .env file.

Laravel uses these values to establish a database connection when interacting with the database. By default, Laravel supports various database systems, including MySQL, PostgreSQL, SQLite, and SQL Server. The DB_CONNECTION variable determines which database driver Laravel should use.

Once you have configured the database connection in the .env file, Laravel will automatically use these settings when performing database operations through its database query builder or Eloquent ORM.

It's important to note that if you make changes to the .env file, you should clear the application cache using the following Artisan command to reflect the updated configuration:

```
php artisan config:cache
```

This command will clear the cached configuration, ensuring that Laravel picks up the latest values from the .env file.

By configuring the database connection in Laravel, you can establish a connection to your desired database system and leverage the database features within your application.

Understand migrations to manage database schema changes.

Migrations in Laravel are a powerful feature that allows you to manage database schema changes in a structured and version-controlled manner. Migrations enable you to modify the database structure, create or alter tables, add or remove columns, and perform other database schema operations.

Here's an overview of how migrations work in Laravel:

Migration Files:

Migrations in Laravel are defined as individual files located in the database/migrations directory of your Laravel project.

Each migration file contains two methods: up and down. The up method defines the actions to be performed when applying the migration, while the down method defines the actions to be performed when rolling back the migration.

Laravel Migration Commands

To view the migration commands, open the terminal, and enter the command

```
"php artisan list"
```

This command lists all the commands available in Laravel

There are six commands of migrate in Laravel:

- migrate:fresh
- migrate:install
- migrate:refresh
- migrate:reset
- migrate:rollback

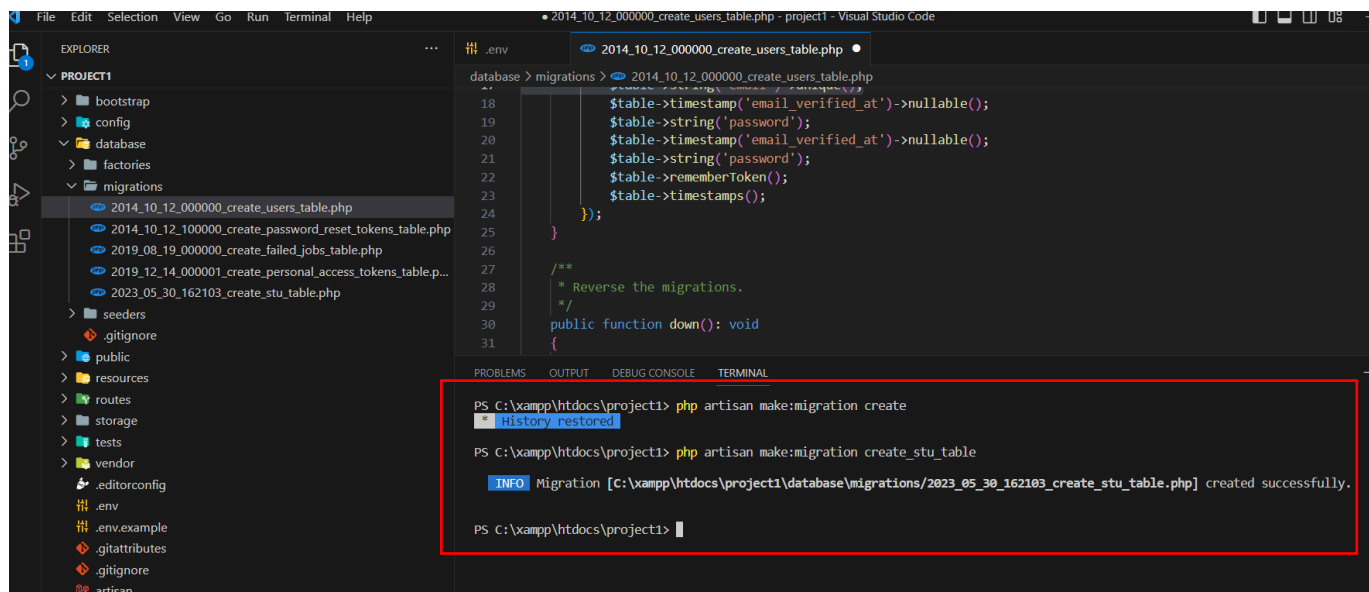
- migrate:status

In Laravel, there are several Artisan commands available to work with migrations. These commands allow you to create migration files, apply migrations, rollback migrations, view migration status, and more. Here are some commonly used Laravel migration commands:

1) Create Migration:

php artisan make:migration create_table_name

This command creates a new migration file in the database/migrations directory with a timestamp prefix and a specified name. The --create option indicates that the migration will create a new table.

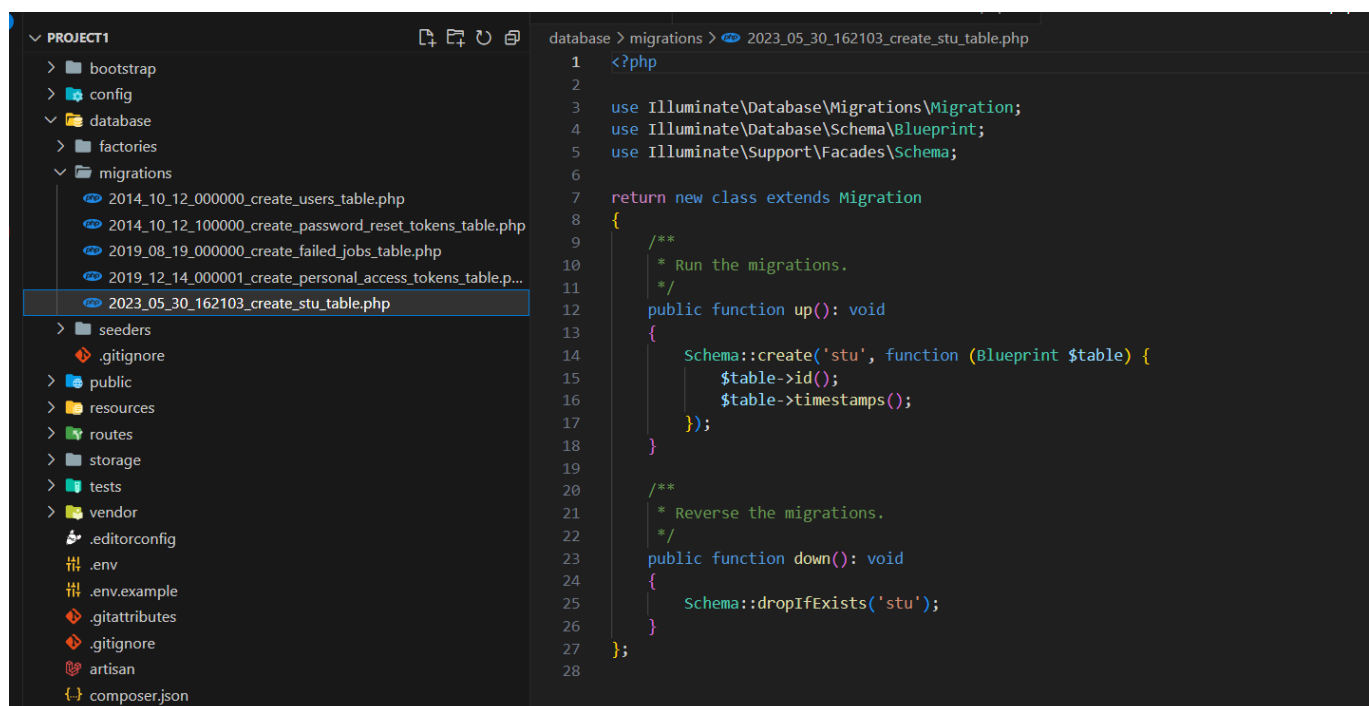


The screenshot shows the Visual Studio Code interface with a Laravel project. The Explorer panel on the left shows the project structure, including the database/migrations directory. The main editor shows the content of the migration file `2014_10_12_000000_create_users_table.php`. The terminal at the bottom shows the command `php artisan make:migration create` being executed, followed by a message indicating that the migration was created successfully.

```
PS C:\xampp\htdocs\project1> php artisan make:migration create
History restored

PS C:\xampp\htdocs\project1> php artisan make:migration create_stu_table
INFO Migration [C:\xampp\htdocs\project1\database\migrations\2023_05_30_162103_create_stu_table.php] created successfully.

PS C:\xampp\htdocs\project1>
```



The screenshot shows the Visual Studio Code interface with a Laravel project. The Explorer panel on the left shows the project structure, including the database/migrations directory. The main editor shows the content of the migration file `2023_05_30_162103_create_stu_table.php`. The code defines a migration class that creates a table named 'stu' with an 'id' column and a 'timestamps' column.

```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('stu', function (Blueprint $table) {
15             $table->id();
16             $table->timestamps();
17         });
18     }
19
20     /**
21      * Reverse the migrations.
22      */
23     public function down(): void
24     {
25         Schema::dropIfExists('stu');
26     }
27 };
28
```

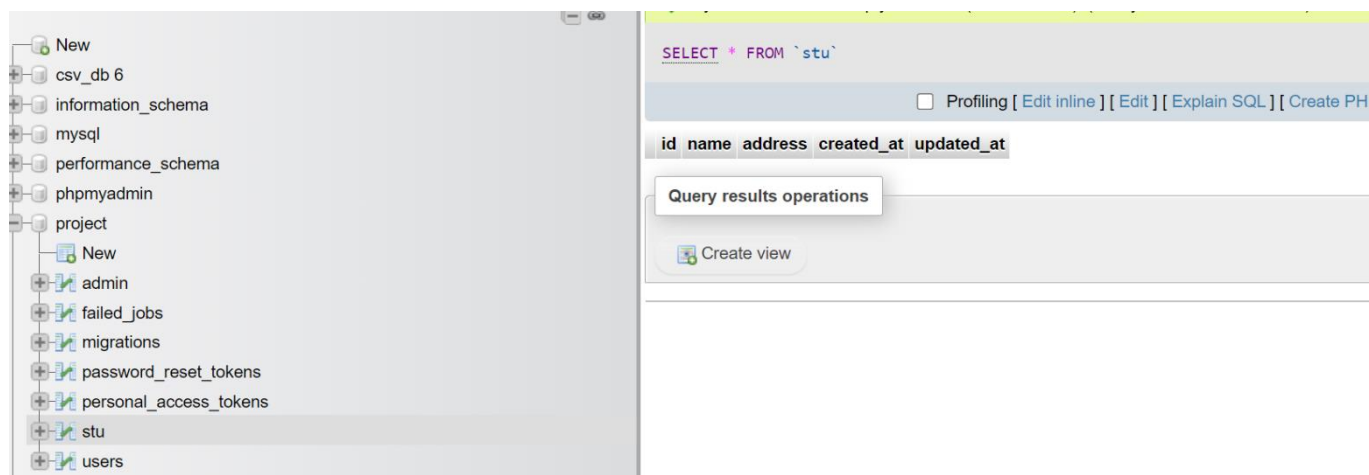
And adding some data do some change in create table file: -

```
database > migrations > 2023_05_30_162103_create_stu_table.php
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  return new class extends Migration
8  {
9      /**
10       * Run the migrations.
11       */
12     public function up(): void
13     {
14         Schema::create('stu', function (Blueprint $table) {
15             $table->id();
16             $table->string('name');
17             $table->string('address');
18             $table->timestamps();
19         });
20     }
21
22     /**
23      * Reverse the migrations.
24      */
25     public function down(): void
26     {
27         Schema::dropIfExists('stu');
28     }
29 };
30
```

Then save it and run the command “php artisan migrate” for saving successful in database

```
PS C:\xampp\htdocs\project1> php artisan migrate
INFO Running migrations.
2023_05_30_162103_create_stu_table ..... 18ms DONE
PS C:\xampp\htdocs\project1>
```

And then for confirmation you also check in MySQL as link manner: -



The screenshot shows the MySQL Workbench interface. On the left, the 'project' database is selected, and the 'stu' table is highlighted in the 'New' section. The main window displays the table structure for 'stu' with columns: id, name, address, created_at, and updated_at. The query 'SELECT * FROM `stu`' is entered in the query editor. Below the query, there are buttons for 'Profiling', 'Edit inline', 'Edit', 'Explain SQL', and 'Create PH'. The 'Query results operations' section shows a 'Create view' button.

2) Create Migration (for Existing Table):

php artisan make:migration add_column_to_table

where--table=table_name

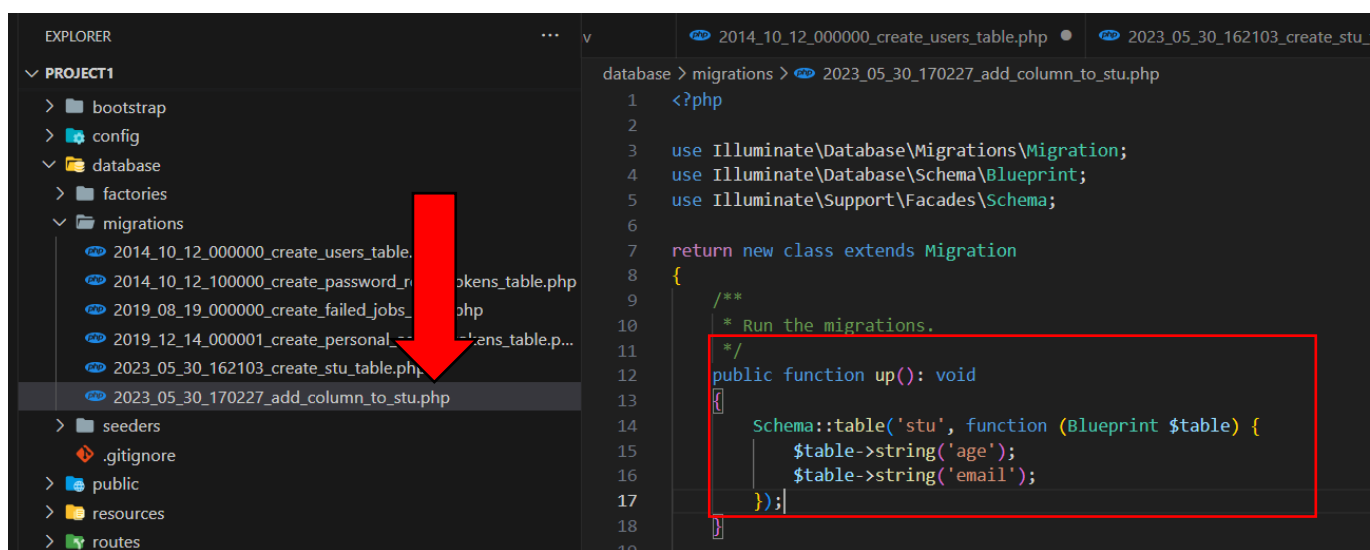
This command generates a new migration file that modifies an existing table. The --table option specifies the name of the table to be modified.

```
PS C:\xampp\htdocs\project1> php artisan make:migration add_column_to_stu

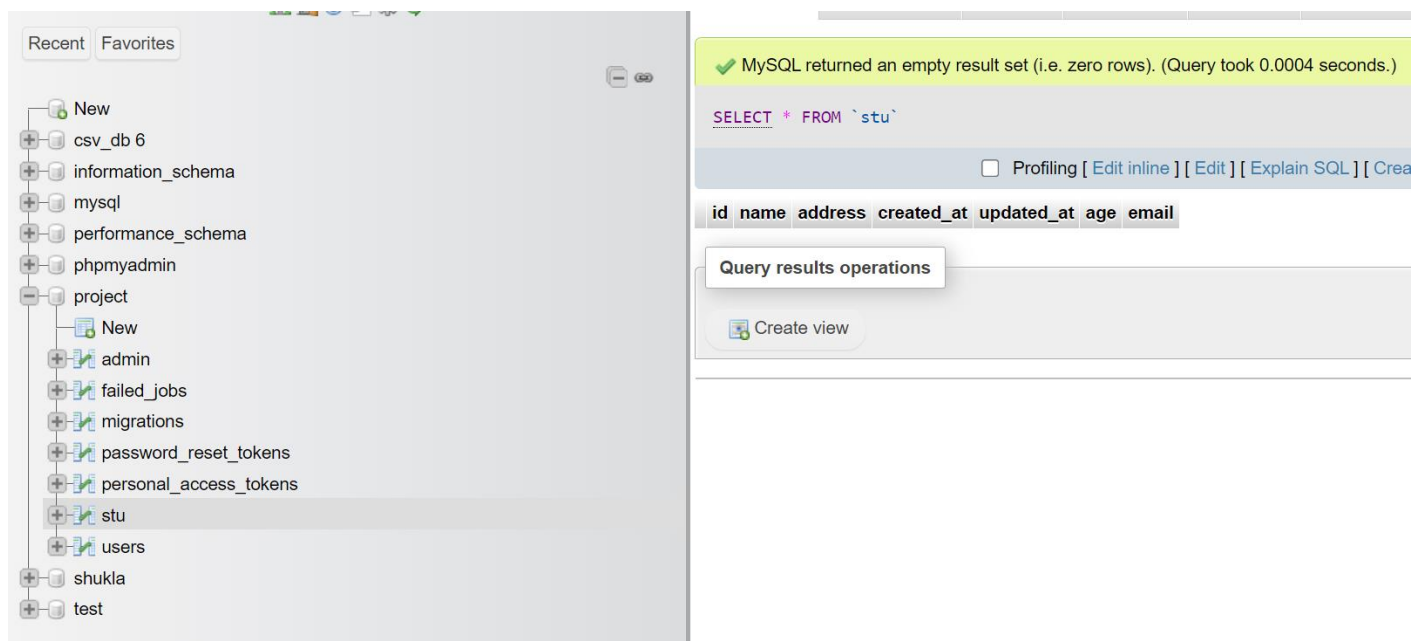
[INFO] Migration [C:\xampp\htdocs\project1\database\Migrations\2023_05_30_170227_add_column_to_stu.php] created successfully

PS C:\xampp\htdocs\project1>
```

You can see in below code the migration file is created and now you can do some change



Then save it and run the command “php artisan migrate” for saving successful in database



3) Run Migrations:

php artisan migrate

This command applies any pending migrations and updates the database schema to the latest version.

4) Rollback Migrations:

php artisan migrate:rollback

This command reverts the last batch of migrations, undoing the changes made to the database schema.

5) Reset Migrations:

php artisan migrate:reset

This command rolls back all migrations, resetting the database schema to its initial state.

6) Refresh Migrations:

php artisan migrate:refresh

This command rolls back all migrations and then re-runs them, effectively resetting and reapplying the migrations.

7) Rollback and Redo a Single Migration:

php artisan migrate:rollback --step=1

php artisan migrate

8) View Migration Status: -

php artisan migrate:status

This command displays the status of each migration, indicating whether it has been run or not

Maintenance Mode

Maintenance mode in Laravel allows you to display a maintenance page or perform maintenance tasks on your application while temporarily taking it offline. It's useful when you need to perform updates, database migrations, or any other tasks that require temporary interruption of the normal application flow.

To enable maintenance mode in Laravel, you can use the down Artisan command. Open your command-line interface, navigate to the root directory of your Laravel project, and run the following command:

```
php artisan down
```

When you run this command, Laravel will create a file called down in the storage directory, indicating that the application is in maintenance mode. By default, Laravel will display a default maintenance page with a 503 HTTP status code

Once you have completed the necessary maintenance tasks, you can bring your application back online by using the up Artisan command:

```
php artisan up
```

Running this command will remove the down file from the storage directory, taking your application out of maintenance mode and allowing normal access to the application.

Maintenance mode is a useful feature in Laravel that helps you gracefully handle application downtime or interruptions, ensuring a smooth user experience during maintenance periods

Eloquent ORM

Eloquent is Laravel's object-relational mapping (ORM) system, which provides an intuitive and expressive way to interact with databases. It allows you to work with database tables as objects and provides methods and relationships to perform various database operations.

Here are some key features and concepts of Eloquent ORM:

1) Model and Database Table Relationship:

In Eloquent, each database table is associated with a corresponding model class. The model class represents a record in the table and provides methods to interact with the data. By convention, the model class is named singular and follows CamelCase naming (e.g., User model for the users table).

2) Retrieving Records:

Eloquent provides a fluent query builder interface to retrieve records from the database. You can use methods like `get()`, `first()`, `find()`, or `where()` to fetch records based on specific conditions. Eloquent also supports eager loading to efficiently retrieve related records.

3) Creating and Updating Records:

Eloquent makes it easy to create and update records. You can create a new record by instantiating a model object and assigning values to its properties, then calling the `save()` method. To update an existing record, retrieve it from the database, modify its properties, and again call the `save()` method.

4) Deleting Records:

Eloquent provides a `delete()` method to remove records from the database. You can call this method on a model instance to delete the corresponding record.

5) Relationships:

Eloquent allows you to define relationships between models, making it easier to work with related data. There are several types of relationships available, including `belongsTo()`, `hasMany()`, `hasOne()`, `belongsToMany()`, and more. These methods define the relationships and provide convenient methods to retrieve related records.

6) Query Scopes:

Eloquent supports query scopes, which are methods defined on a model to encapsulate reusable query conditions. Scopes allow you to define common query constraints and apply them to queries effortlessly.

7) Accessors and Mutators:

Eloquent provides accessors and mutators to define custom getters and setters for model attributes. This allows you to perform transformations or manipulate data before retrieving or storing it in the database.

8) Mass Assignment Protection:

Eloquent offers built-in protection against mass assignment vulnerabilities. You can specify the fillable or guarded attributes on a model to control which attributes can be mass assigned when creating or updating records.

9) Events:

Eloquent triggers events during various stages of a model's lifecycle, such as creating, updating, deleting, or retrieving records. You can define event listeners to perform additional actions or execute custom logic when these events occur.

What is model :-

In Laravel, a "model" refers to a class that represents a specific table in a database. Models in Laravel are used to interact with the database, retrieve data, perform data manipulation, and define relationships between different tables.

Models provide an object-oriented way of working with the database, allowing you to perform database operations using methods and properties defined within the model class. By using models, you can abstract away the direct interaction with the database and work with data in a more intuitive and expressive manner.

Here are some reasons why we use models in Laravel:

- 1) **Data Abstraction:** Models provide a layer of abstraction between your application code and the database. They encapsulate the logic for retrieving and manipulating data, allowing you to work with data as objects rather than writing raw SQL queries.
- 2) **Object-Relational Mapping (ORM):** Laravel's Eloquent ORM is an implementation of the Active Record pattern, where each database table has a corresponding model class. Models enable you to define relationships between tables using methods and properties, making it easy to work with related data.
- 3) **Query Building:** Models provide a fluent query builder interface, allowing you to build complex database queries using chainable methods. This makes it easier to construct queries with conditions, joins, sorting, and pagination.
- 4) **Validation and Mass Assignment:** Models in Laravel include features for data validation and mass assignment protection. You can define rules for validating input data before saving it to the database, ensuring data integrity. Mass assignment protection helps prevent unexpected data changes by specifying which attributes can be mass assigned.
- 5) **Event Hooks:** Laravel models allow you to define event hooks that are triggered during various stages of a model's lifecycle, such as creating, updating, and deleting records. These hooks enable you to perform additional actions or execute custom logic when specific events occur.

By utilizing models in Laravel, you can streamline database operations, improve code organization, enhance data integrity, and simplify working with related data in your application.

How to Create model in Laravel:-

To create a model in Laravel, you can use the Artisan command-line tool provided by Laravel. Here's the step-by-step process:

Open your terminal or command prompt and navigate to the root directory of your Laravel project.

Run the following command to create a model:

```
php artisan make:model ModelName
```

- If you would like to generate a database migration when you generate the model, you may use

the --migration or -m option:

```
php artisan make:model User --migration
```

Laravel will generate a new file under the app directory with the specified model name. By default, the file will be created in the app namespace.

1) Model connection with database

2) Crud Operations

Data insert

Data fetch

Data update

Data Delete

Model Connection with Database table:-

Data insert

Step 1:- create a form using post method for access data from user

Step2:- create a controller for form validation and access the data in view

Step3:- set the route path for form view

Step4 :- now create a table or migration for data enter in database and edit the column and save it

Step5:- now create a model for migration and do some commands

Like :

Protected table

Protected id

Step 6: - after save model file then change in controller

1) Add model in controller file: -

2) set some command for inserting form data in database table: -

```
$admin= new Admin; (where Admin is Model name and admin is model class object)
```

```
$admin->name=$ab->name;
```

```
$admin->mail=$ab->mail;
```

```
echo $admin->save ();
```

and then save it

Activity:- Create a Form which inserts the data in the database

Activity :- Create a Model and Demonstrate all CRUD operation

Select data from database table: -

Step1:- for fetch data we have to need a page where we can show fetch data

Step2:- for see data need to define route

Step3:- now open controller then create function

Step 4:- now submit the details in the form then you easily see the result

Delete Operations:-

Step 1:- create a button in views file which you use for fetching data

Step 2:- Go to the controller file and create a function.

Step 3:- Now go the web.php and create a route for delete.

Step 4:- Now to the form-view.blade.php file and give the route to the Delete Button.

Step 5:- Now if you click on the delete button you will see that your data will be deleted.

Update Operations:-

Step 1:- Now again go to the Form-view file and give the route to the “edit” button.

Step 2:- Go to the web.php file and give the route for edit

Step 3:- Now go to the controller file and create a function.

Step 4:- Again go to the web.php file and now give the route for update using post method.

Step 5:- In the same controller file go to the Index function and do some changes.

Step 6:- Now go to the form.blade.php file and provide the url in form action and the title also.

Step 7:- In the same file you have to provide the value of the data which you want to update.

Step 8: - Go to the Controller file and create a function to update.

Step 9:- Now you are able to Create, Read, Update and Delete data in Laravel.

Mass Assignment:-

In Laravel, mass assignment refers to the ability to quickly and conveniently assign values to multiple model attributes in a single statement. It allows you to set multiple attributes of a model by passing an array of values, which can be useful when working with forms or handling incoming data.

To enable mass assignment in Laravel, you need to define the fillable or guarded attributes on your model.

Define the fillable or guarded attributes in your model:

Fillable: Specify the attributes that are allowed to be mass assigned.

Guarded: Specify the attributes that are not allowed to be mass assigned.

Here's an example using the fillable approach:

Here's an example using the fillable approach:

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class Example extends Model  
{
```

```
protected $fillable = ['field1', 'field2', 'field3'];  
}
```

In this example, field1, field2, and field3 are the attributes that can be mass assigned.

Defining Eloquent Relationships:-

In Laravel, Eloquent is the ORM (Object-Relational Mapping) system that comes with the framework. It provides a convenient and expressive way to define and work with database relationships. Eloquent relationships allow you to define how different database tables or models are related to each other.

Laravel supports several types of relationships, including:

One-to-One Relationship:

In a one-to-one relationship, each record in one table is associated with exactly one record in another table.

You can define a one-to-one relationship using the `hasOne` and `belongsTo` methods.

One-to-Many Relationship:

In a one-to-many relationship, a single record in one table can be associated with multiple records in another table.

You can define a one-to-many relationship using the `hasMany` and `belongsTo` methods.

Many-to-Many Relationship:

In a many-to-many relationship, multiple records in one table can be associated with multiple records in another table.

You can define a many-to-many relationship using the `belongsToMany` method.

Has-Many-Through Relationship:

In a has-many-through relationship, a model can access distant relatives via intermediate models.

You can define a has-many-through relationship using the `hasManyThrough` method.

Polymorphic Relationships:

Polymorphic relationships allow a model to belong to more than one type of model on a single association.

You can define a polymorphic relationship using the `morphTo`, `morphOne`, `morphMany`, and `morphToMany` methods.

To define relationships in Laravel, you need to set up the appropriate methods and properties in your Eloquent model classes. Here's an example of how you can define a one-to-many relationship between a User model and a Post model..

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
```

```
{
```

```
    public function posts()
```

```
    {
```

```
        return $this->hasMany(Post::class);
```

```
    }
```

```
}
```

```
class Post extends Model
```

```
{
```

```
    public function user()
```

```
    {
```

```
        return $this->belongsTo(User::class);
```

```
    }
```

```
}
```

In this example, the User model has a `hasMany` relationship with the Post model, and the Post model has a `belongsTo` relationship with the User model. You can then use these relationships to perform various operations, such as retrieving

related records, creating new records with associations, or querying based on relationships.

For example, to retrieve all posts of a user, you can do:

Live wire in Laravel:-

In Laravel, "Live Wire" refers to a full-stack framework for building dynamic web applications using server-side rendering (SSR) techniques. It allows you to create rich user interfaces with the convenience and simplicity of Laravel's backend development.

Live Wire enables you to build interactive UI components in Laravel using a combination of PHP, HTML, and JavaScript. It utilizes a real-time connection between the server and the browser, allowing seamless updates to the UI without requiring a full page reload.

Here's a brief overview of how Live Wire works:

Component Creation: You define your UI components as PHP classes in Laravel. These components contain the logic and HTML structure of the UI elements you want to render.

Data Binding: You can bind data properties to the UI elements within your components. These properties can be updated on the server-side, and Live Wire automatically synchronizes the changes with the client-side UI.

Event Handling: Live Wire provides a simple syntax for handling user interactions and events within your components. You can define methods that respond to user actions like button clicks, form submissions, and more.

AJAX Requests: When an event occurs, Live Wire intercepts the action and sends it to the server via AJAX. The server-side code processes the action, updates the data properties, and returns the updated HTML for the component.

Re-rendering: Upon receiving the updated HTML, Live Wire re-renders the component on the client-side, applying the changes to the UI without refreshing the entire page. This gives the illusion of a real-time and reactive user experience.

Live Wire simplifies the development of dynamic interfaces in Laravel by abstracting away the complexities of managing the server-client communication. It provides a smooth and intuitive workflow for building interactive web applications within the Laravel ecosystem.

Difference between livewire vs jetsream:-

Live Wire and Jetstream are two separate packages in the Laravel ecosystem, serving different purposes:

Live Wire: Live Wire is a full-stack framework for building dynamic web interfaces using server-side rendering (SSR) techniques. It focuses on providing a seamless way to build interactive UI components within Laravel using a combination of PHP, HTML, and JavaScript. Live Wire enables real-time updates to the UI without requiring a full page reload. It simplifies the development of dynamic interfaces and is suitable for building feature-rich user interfaces within a Laravel application.

Jetstream: Jetstream, on the other hand, is an application scaffolding package that provides a starting point for building modern web applications. It offers a robust foundation for user authentication, authorization, and management. Jetstream includes features like user registration, login, two-factor authentication (2FA), API support, team management, and profile management. It also provides a customizable front-end UI powered by Tailwind CSS. Jetstream is designed to accelerate the development process and provide a secure and scalable authentication and user management system.

Installation of live wire:-

To install Live Wire in your Laravel application, you can follow these steps:

Set up a Laravel Project: If you don't have a Laravel project yet, you'll need to create one. You can do this using Composer by running the following command in your terminal:

Step 1:- composer create-project project name

Step2:- Install Live Wire: Once you have a Laravel project set up, navigate to your project's directory and use Composer to install Live Wire by running the following command:

composer require livewire/livewire

Step3:- Publish Assets (Optional): You can publish the Live Wire assets, including configuration files and views, to your project for customization. Run the following command in your terminal:

php artisan livewire: publish

step 4:- create a blade file in views folder and add all these syntax:-

step 5:- Include the JavaScript and CSS: Add the following lines to your application's layout or view file to include the necessary JavaScript and CSS files for Live Wire:

```
<head>
    <!-- ... -->
    @livewireStyles
</head>
<body>
    <!-- ... -->
    @livewireScripts
</body>
```

Step 6:- Start Building Components: You are now ready to start building Live Wire components. Create a new PHP class that extends the Livewire\Component base class. This class will represent your component and contain its logic and UI

structure. For example, create a HelloWorld component by running the following command:

```
<div>  
  <h1>Welcome to my Live Wire Application!</h1>  
</div>
```

Step 7:- now add the components file in blade file & save it...

Step 8:- Serve the Application: Run the development server using the following command:

```
php artisan serve
```