Learning Outcome

## 

## Laravel Framework

## (15 Hrs)

## 

# Migrations in Laravel

## What is Laravel Migration?

Laravel migration is a process that enables you to build a table in a database without generally writing or modifying the SQL queries, It allows you to roll back changes, make incremental changes or sync the database structure simultaneously when multiple teams work on the same application.

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

**Why do we need Laravel Migration?**

Suppose we are working in a team, and some idea strikes that require the alteration in a table. In such a case, the SQL file needs to be passed around, and some team member has to import that file, but team member forgot to import the SQL file. In this case, the application will not work properly, to avoid such situation, Laravel Migration comes into existence.

Laravel Migration allows you to add a new column or delete the records in your database without deleting the records that are already present.

## Laravel Migration Commands

To view the migration commands, open the Git bash window, and enter the command "**php artisan list**". This command lists all the commands available in Laravel.

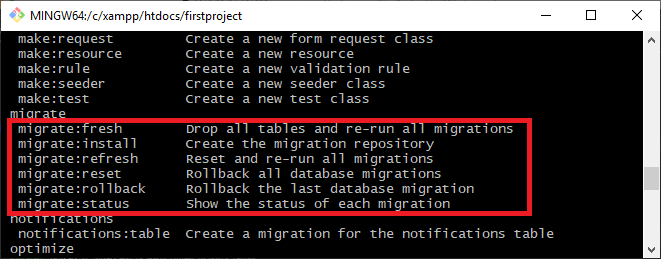


Image 1: Laravel Migration Command

There are six commands of migrate in Laravel:

* migrate:fresh
* migrate:install
* migrate:refresh
* migrate:reset
* migrate:rollback
* migrate:status

**migrate:fresh**

Using this command, one can drop all the tables from the database and re-run all the migrations

**Syntax**: php artisan migrate:fresh

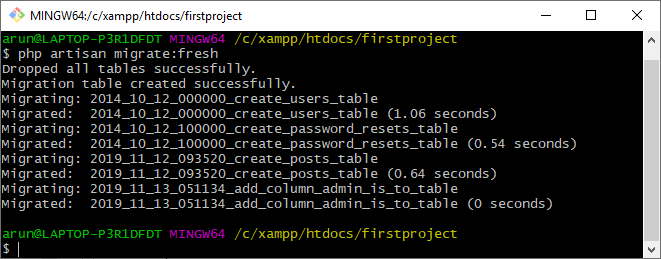


Image 2: migrate:fresh

**migrate:install**

Using this command, a migration table can be created in a database.

**Syntax**: php artisan migrate:install

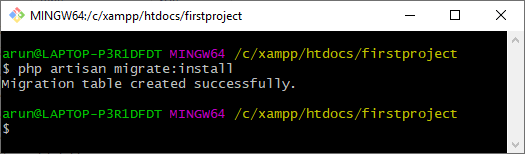


Image 3: migrate:install

**migrate:refresh**

Using this command, all the migrations can be rolled back, and migrations can be re-run.

In simple words, the database can be re-created entirely.

**Syntax**: php artisan migrate:refresh

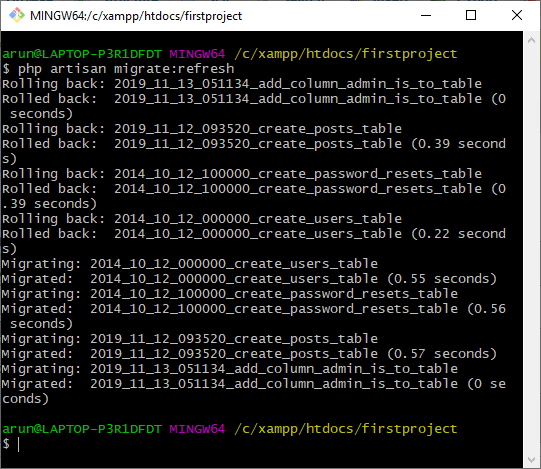


Image 4: migrate:refresh

**migrate:reset**

It solely drops all the tables that you have created in a database.

Syntax: php artisan migrate:reset

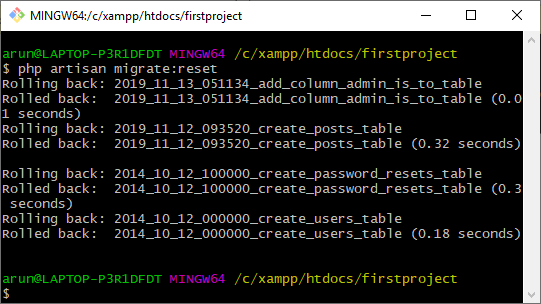


Image 5: migrate:reset

**migrate:rollback**

This Laravel command is used to drop the last database migration.

**Syntax:** php artisan migrate:rollback

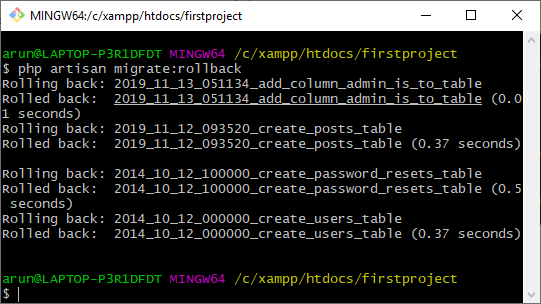


Image 6: migrate:rollback

**migrate:status**

This command helps you check the migration status.

**Syntax**: php artisan migrate:status

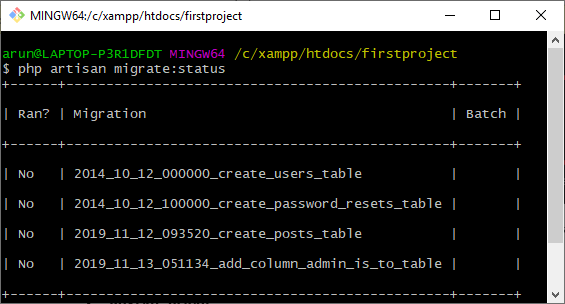


Image 7: migrate:status

## Migrate Specific Migration File in Laravel 8

When you create migrations in laravel, you also have command to migrate it and create tables schemas in database. Migration file is the file which creates the table structure.

When we use migrate command from artisan command list, it will run all migrations and create table. But in some cases, we want to run only a specific migration instead to run all.

**Here is the command to create a laravel project-**

**$ composer create-project --prefer-dist laravel/laravel blog**

**To start the development server of Laravel**

$ php artisan serve

URL: http://127.0.0.1:8000

Assuming Laravel already installed inside your system.

**Create Database & Connect**

To create a database, either we can create via Manual tool of PhpMyAdmin or by means of a mysql command.

**CREATE DATABASE laravel\_app;**

To connect database with application, Open .env file from application root. Search for DB\_ and update your details.

* DB\_CONNECTION=mysql
* DB\_HOST=127.0.0.1
* DB\_PORT=3306
* DB\_DATABASE=laravel\_app
* DB\_USERNAME=root
* DB\_PASSWORD=root

**Create Migration**

By default, after fresh installation of Laravel, when you will see into /database/migrations folder you should see 3 default migration files.

To create new migration file, here is the artisan command.

Open project into terminal and run this command.

**$ php artisan make:migration create\_products\_table**

It will create a migration file with name like 2021\_05\_01\_092040\_create\_products\_table.php inside /database/migrations folder.

Open migration file and write this complete code.

## 

Image 8: Migration Code

Every migration we create extends the Migration class, and must contains two methods “up” and “down”

* up() – The up () method includes the set of operation that is executed when the migration is run.
* down() – The down () method includes the set of operations that is executed when you rollback migration , it usually reverse the operations performed by the up () method.

**Migrate Migrations in Laravel**

We have artisan command to migrate Laravel migration files.

**Migrate All Migrations**

**$** **php artisan migrate**

It will migrate all migrations from /database/migrations folder create their respective tables.

**Migrate a Specific Migration File**

$ php artisan migrate:refresh --path=database/migrations/2021\_05\_01\_092040\_create\_products\_table.php

Syntax – $ php artisan migrate:refresh –path=<MIGRATION FILE PATH>

Inside this case, migration file path is database/migrations/2021\_05\_01\_092040\_create\_products\_table.php

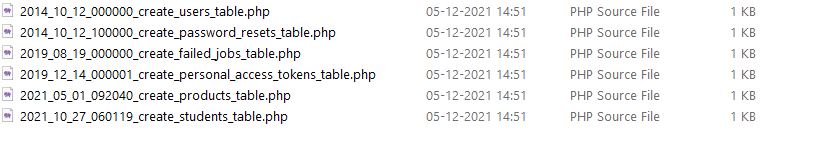


Image 9: Migration Files (Folder)

# Database Seed in Laravel

## What is Database Seeding?

Database seeding means fill your database tables with the test data programmatically. With this approach, one can enter thousands of records in the database within a few minutes. It is an efficient way rather than doing it manually. You may want a dummy data to test out your application and seeding the database through the program saves you a ton of time.

Laravel provides a tool to add sample or dummy data to our databases automatically. which calls it database seeding.

Laravel includes the ability to seed your database with test data using seed classes. All seed classes are stored in the **database/seeders** directory. By default, a Database Seeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order

## Create Database Seeder

To create database seeder, we will below steps

* Install Laravel 8
* Create Seeder
* Write Code in Seeder
* Run the Seeder

**Step – 1 Installation of Laravel Application**

Run the below command to install Laravel

**composer create-project --prefer-dist laravel/laravel learn**

**Step – 2 Create Seeder in Laravel**

Run the below command to create seeder in Laravel, this Below command will create a seeder under **database/seeders** named as **UsersTableDataSeeder.php**

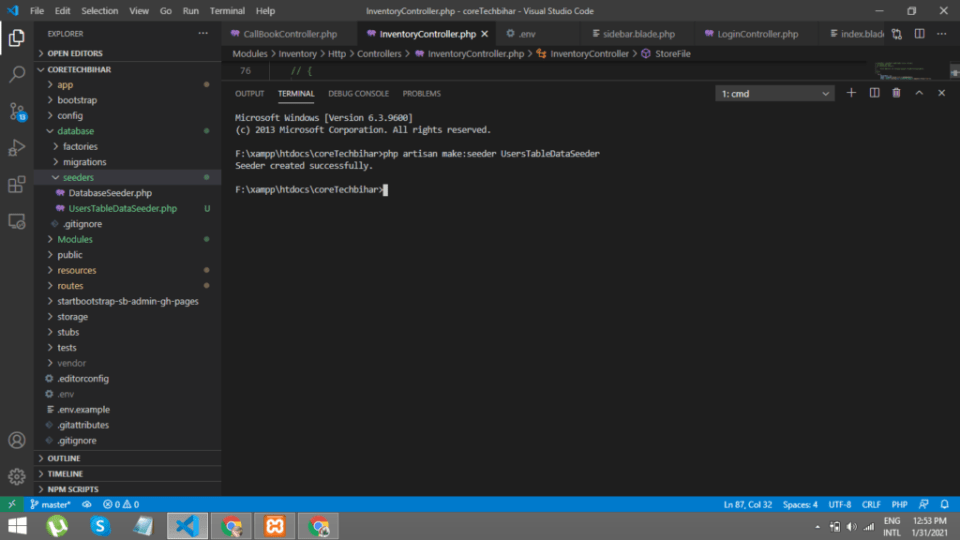
****

Image 10: Create Seeder

**Step – 3 Write Code in Seeder**

Now below code will create dummy data for users table, I will insert username, email and password.

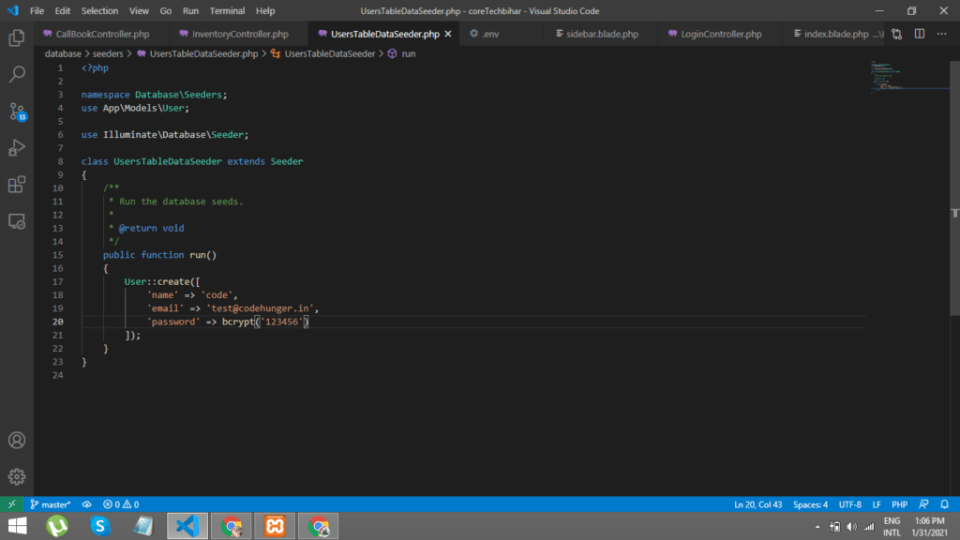


Image 11: Write Seeder

**Step – 4 Run the seeder**

**php artisan db:seed --class=UsersTableDataSeeder**

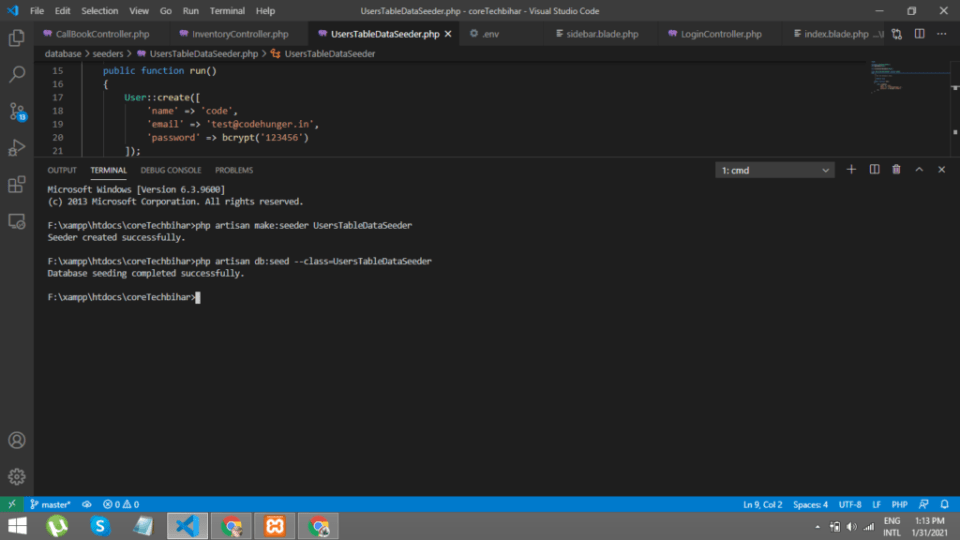


Image 12: Run the Seeder

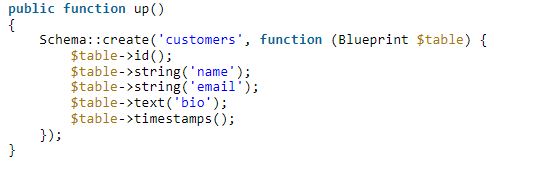
Database Seeding

I am going to create a ‘customers’ table with the columns name, email, and bio. We will see the process of seeding the single table. In the same way, users can seed as many tables as they want.

Create a migration for customers table using the command:

**php artisan make:migration create\_customers\_table**

Open the migration file and add the columns to the ‘up’ method as shown below.



Run the migrate command which will create a ‘customers’ table in your database.

**php artisan migrate**

Next, create a model ‘Customer’ which maps to the ‘customers’ table. Starting Laravel 8.x your models will be created inside the ‘App/Models’ directory. For this tutorial, I assume you are using Laravel 8.x. In the older version of Laravel, you need to adjust the model path.

**php artisan make:model Customer**

## Seeding a Database

Seeding a database in Laravel requires you to follow a few steps which are writing seeders, creating a model factory and running seeders. Let’s do it one by one.

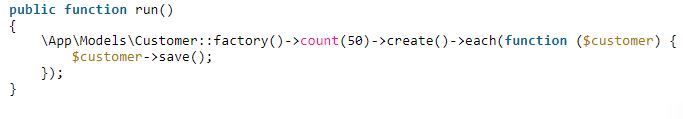
**Writing Seeders**

If you look at the Laravel filesystem then you notice the directory **database/seeders**. All the seed classes will go inside this directory. A seed class has the run method from where you can insert the data in the table.

Make a seed class CustomerSeeder using the command:

**php artisan make:seeder CustomerSeeder**

After executing this command, you should see **CustomerSeeder.php** file created under **database/seeders**. Open this file and to the run function add the below code.



Here I am calling the Customer factory which we will create in the next step. Notice I passed the value as 50 to the ‘count’ method. It means 50 rows should be inserted in the ‘customers’ table. Adjust this value as you wish.

**Creating Factories**

The code I have written in CustomerSeeder will look for a factory to get the test data. It means we have to create a model factory. I will give a name to the factory as CustomerFactory. Run the command to generate the model factory.

**php artisan make:factory CustomerFactory --model=Customer**

This command will create **a CustomerFactory.php** inside the **database/factories** directory. Open this file and to the definition method generate a test data as follows.

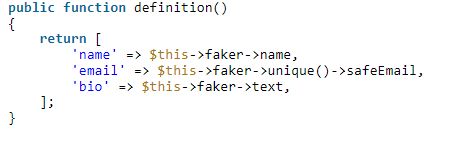


Image 13: Creating Factories

The above array returns a randomly picked name, email, and text for bio.

**Running Seeders**

We all set with the seeder and factory. Now run our seeder using the command below.

**php artisan db:seed --class=CustomerSeeder**

Check your ‘customers’ table and it should have the 50 dummy entries inserted.

The above command runs a single seeder which we passed explicitly. Let’s say you want to create multiple seeders and run all of them using a single command. In such cases, you should call your seeders from the **DatabaseSeeder**which is found in the same directory **database/seeders.**

The user can create as many seeders as per their requirements and call them from this run method.

Finally, run the seeders using the command below.

**php artisan db:seed**

It will call the all seeders from the run method and execute it.

# Handing Session in Controller and Views

## Laravel session

Laravel session is a way of storing the user information across the multiple user requests. It keeps track of all the users that visit the application.

Let's understand the session through an example.

First, we create a form on which we apply the properties of the session.

**form.blade.php**



Image 14: form.blade.php

Now, we define the store() function in FormController.php file.

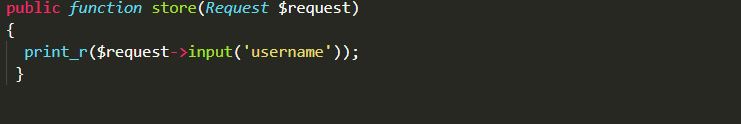


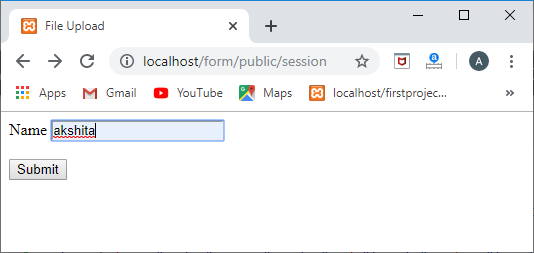
Image 15: FormController.php

At the end, we define the route in web.php.

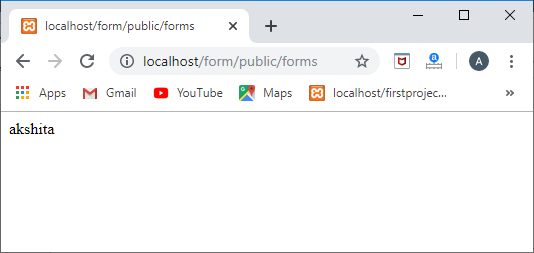


Image 16: web.php

**Output**



When we click on the submit button, then the screen appears which is shown below:



### Storing the data in a session

To store the user name in a session, we use the put() method of session as shown below:

**$request->session()->put('user', $request->input('username'));**

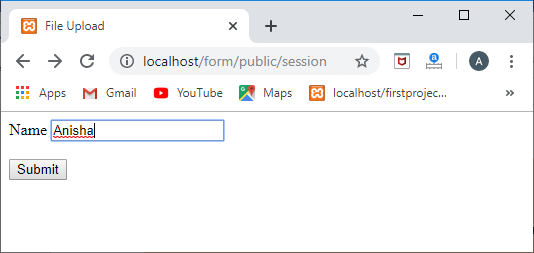
To retrieve the session, we use the get() method of session as shown below:

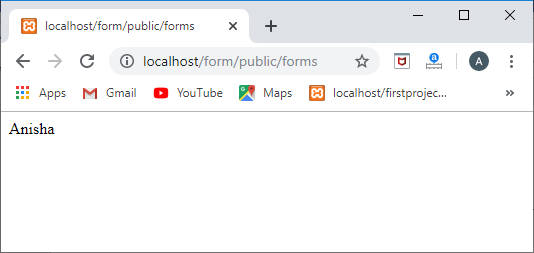
**echo $request->session()->get('user');**



Image 17: Storing data in Session

**Output**





### Global Session Helper

We can also use the global session function that stores and retrieves the value in a session. When the session function is passed with a single parameter, then it returns the value of the key. If the session is passed with an array of key/value pairs, then the values are stored in the session.

// Retrieve a data from the session key.

**$data=session('key');**

//Providing a default value to the session key.

**$data=session('key', 'default');**

// Storing the value in the session key.

**session(['key'=>'value']);**

Let's understand through an example.

**FormController.php**

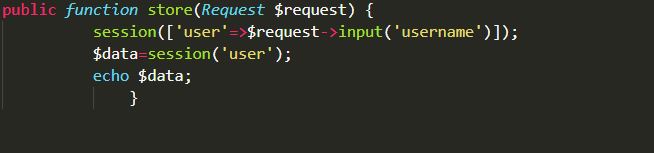


Image 18: FormController.php

### Retrieving all session data

If we want to retrieve all the session data, then we can use the **all()** method as shown below:

**$session\_data = $request->session()->all();**

Let's understand through an example:

**FormController.php**

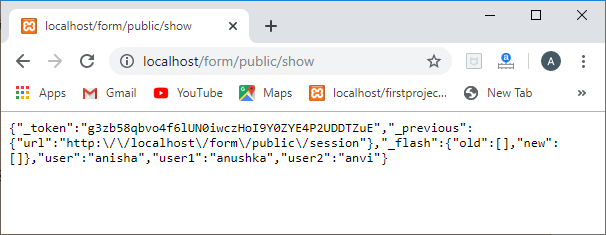


Image 19: FormController.php

Now, we define the route in web.php file.

**Route::get('/show','FormController@store');**

**Output**



### Deleting the session

Now, we will see how to delete the data from the session. We can delete the session by using the forget() method.

Let's understand through an example.

FormController.php

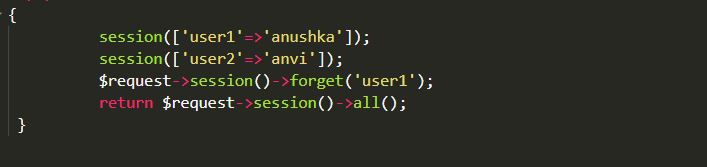
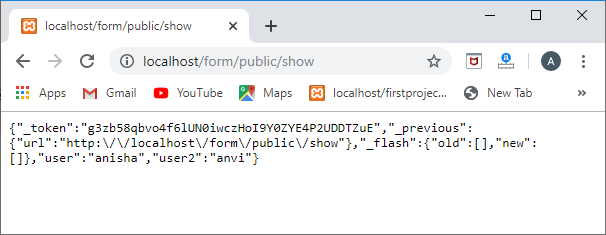


Image 20: FormController.php



In the above screenshot, we can see that the user1 is not displayed, so it means that the user1 has been deleted from the session.

**To remove all the data from the session, then we will use the flush() method.**

**$request->session()->flush();**

Let's understand the flush() method through an example.

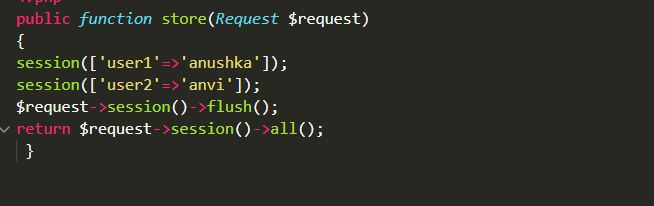
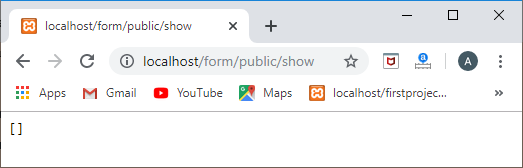


Image 21: Flush

**Output**



In the above screenshot, we observe that all the data has been removed from the session, and it returns an empty array.

**Flashing data**

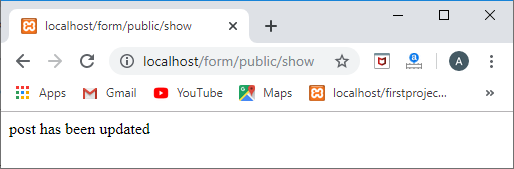
Flash data is useful when we want to store the data in the session for the current request as the flashed data is removed in the next request.

Let's understand flashing data through an example.

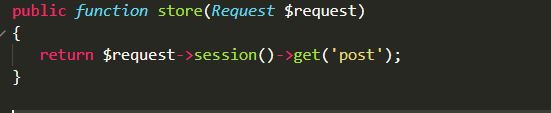


Image 22: Flashing

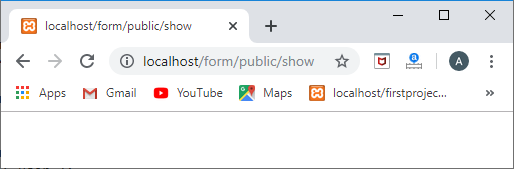
**Output**



When we remove the flash() function from the code, then the code would look like:



When we refresh the page twice, then on the second refresh, the session data will be deleted.



**Note:**  
If you want to keep the flash data for several requests, then we use the reflash() method.  
session()->reflash();  
If you want to keep the specific data for several requests, then we use the keep() method.  
$request->session()->keep('message');

# .env File and its usage

## Environment Configuration

Environment variables are those which provide a list of web services to your web application. All the environment variables are declared in the .env file which includes the parameters required for initializing the configuration.

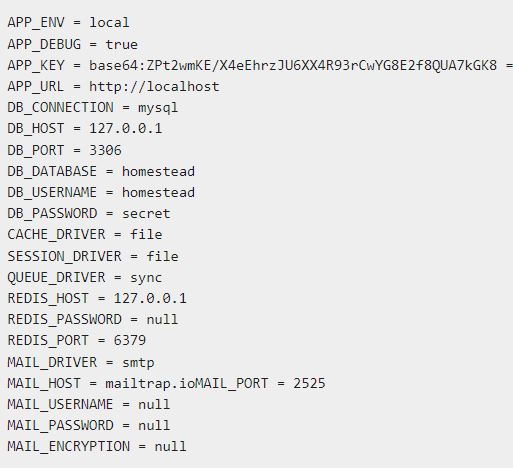


Image 23: .env File

* The .env file should not be committed to the application source control, since each developer or user has some predefined environment configuration for the web application.
* For backup options, the development team should include the .env.example file, which should contain the default configuration.

## What is .env file in Laravel 8?

When you are developing a web app you often have different environments to test your web app. For example:

* Local or Development
* Staging
* Production

You may also have different developers working on the same app and they might need to set their environment differently than others. Therefore, Laravel provides you with .env file that works across different environments.

**Note:** .env file should not be committed to git project.

Laravel's default .env file contains some common configuration values that may differ based on whether your application is running locally or on a production web server.

Laravel also provides you. env.example file where you can only put placeholder keys so that different developer or environment can use different values according to their need.

Do not add sensitive information either in. env or. env.example file that would create security issue and some hacker can easily sniff this type of data from your repository.

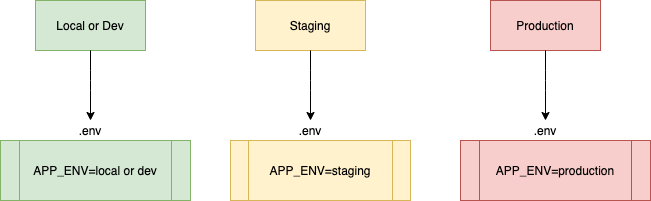
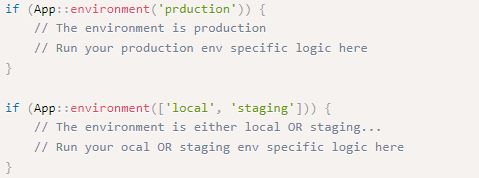


Image 24: .env

You have three different web environments to test your app. For example: dev/local, staging or production. Each of your environment is using different .env file because we do not commit .env file you have to create this file manually on each environment.

Now, we have set our APP\_ENV variable on each environment with different values. Say you want to run certain logic when env is either staging or local you can use following Laravel function to check the environment.



## What are configuration files in Laravel 8?

These configuration files allow you to configure things like your database connection information, your mail server information, as well as various other core configuration values.

All of the configuration files for the Laravel framework are stored in the config directory.

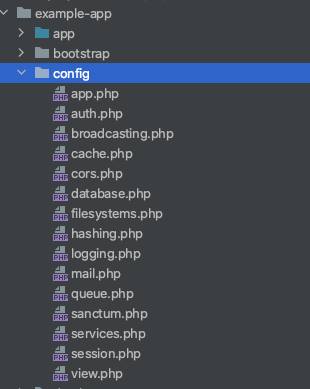


Image 25: Configuration Files

Configuration file uses. env variables to fetch the configuration dynamically. For example: your database configuration might be different on local, staging or on production server depending on what is stored inside your .env file in each environment.

If you open config/database.php file it might look like following:

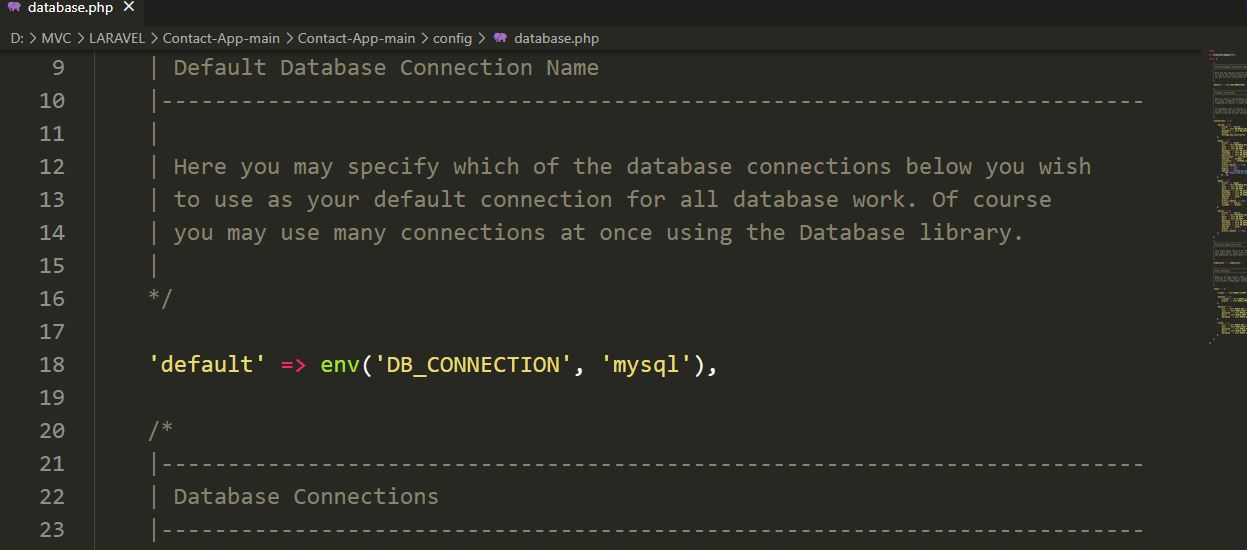


Image 26: database.php

## What is the use of .ENV file?

env file. It's actually a simple configuration text file that is used to define some variables you want to pass into your application's environment. This file needs a something like a parser to make it work. The parser reads the variable definitions one-by-one and parses them to the environment.

. env file, as its name suggest, is a local where you put all your environment setup, such as database credentials, cache drivers and etc. Everything that is about the server that the project is running, and may have different values for different servers, are setup here.

**Uses:**

Environment File Security

Your .env file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

Additional Environment Files

Before loading your application's environment variables, Laravel determines if either the APP\_ENV environment variable has been externally provided or if the --env CLI argument has been specified. If so, Laravel will attempt to load an. env. [APP\_ENV] file if it exists. If it does not exist, the default .env file will be loaded.

Debug Mode

The debug option in your config/app.php configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the APP\_DEBUG environment variable, which is stored in your .env file.

For local development, you should set the APP\_DEBUG environment variable to true. In your production environment, this value should always be false. If the variable is set to true in production, you risk exposing sensitive configuration values to your application's end users.

**Retrieval of Environment Variables**

All the environment variables declared in the .env file can be accessed by env-helper functions which will call the respective parameter. These variables are also listed into $\_ENV global variable whenever application receives a request from the user end. You can access the environment variable as shown below

**'env' => env('APP\_ENV', 'production'),**

env-helper functions are called in the app.php file included in the config folder. The above given example is calling for the basic local parameter.

**Accessing Configuration Values**

You can easily access the configuration values anywhere in the application using the global config helper function. In case if the configuration values are not initialized, default values are returned.

For example, to set the default time zone, the following code is used

**config(['app.timezone' => 'Asia/Kolkata']);**

**Caching of Configuration**

To increase the performance and to boost the web application, it is important to cache all the configuration values. The command for caching the configuration values is

**php artisan config:cache**

**Maintenance Mode**

Sometimes you may need to update some configuration values or perform maintenance on your website. In such cases, keeping it in maintenance mode, makes it easier for you. Such web applications which are kept in maintenance mode, throw an exception namely MaintenanceModeException with a status code of 503.

You can enable the maintenance mode on your Laravel web application using the following command

**php artisan down**

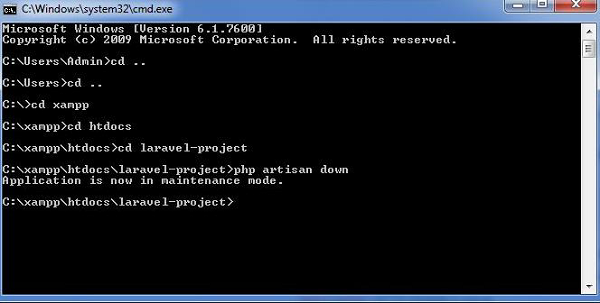


Image 27: down

Once you finish working on updates and other maintenance, you can disable the maintenance mode on your web application using following command

**php artisan up**

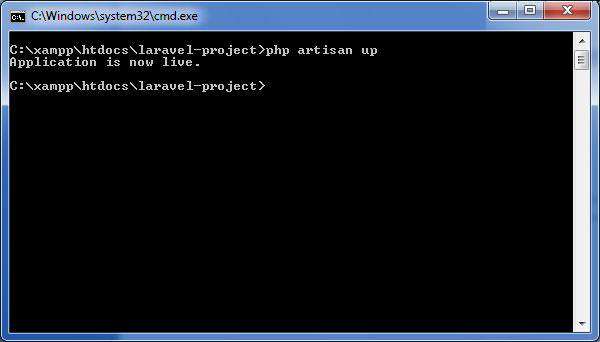


Image 28: up

# Introduction to JetBrains (Livewire)

## Laravel Jetstream

Laravel Jetstream is a beautifully designed application starter kit for Laravel and provides the perfect starting point for your next Laravel application. Jetstream provides the implementation for your application's login, registration, email verification, two-factor authentication, session management, API via Laravel Sanctum, and optional team management features.

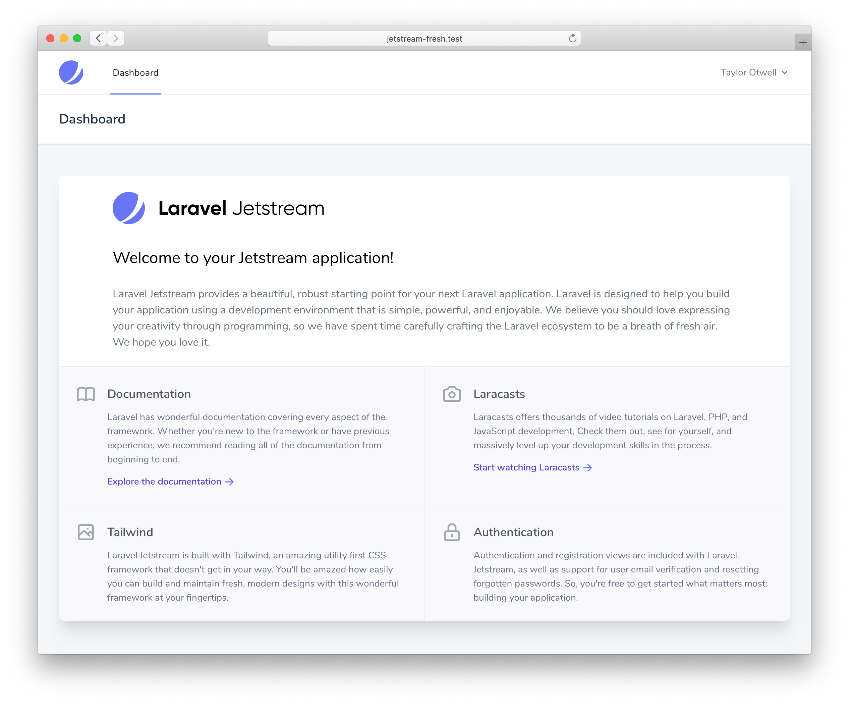
I

Image 29: Laravel Jetstream

## Laravel Livewire (Livewire + Blade)

Laravel Livewire is a library that makes it simple to build modern, reactive, dynamic interfaces using Laravel Blade as your templating language. This is a great stack to choose if you want to build an application that is dynamic and reactive but don't feel comfortable jumping into a full JavaScript framework like Vue.js.

When using Livewire, your application's routes will respond with typical Blade templates. However, within these templates you may render Livewire components as necessary:



**Components**

When we created the Jetstream Livewire stack, a variety of Blade components (buttons, panels, inputs, modals) were created to assist in creating UI consistency and ease of use. You are free to use or not use these components. However, if you would like to use them, you should publish them using the Artisan vendor:publish command:

**php artisan vendor:publish --tag=jetstream-views**

You may gain insight into how to use these components by reviewing their usage within Jetstream's existing views located within your application's resources/views directory

**Modals**

Most of the Jetstream Livewire stack's components have no communication with your backend. However, the Livewire modal components included with Jetstream do interact with your Livewire backend to determine their open / closed state.

In addition, Jetstream includes two types of modals: dialog-modal and confirmation-modal. The confirmation-modal may be used when confirming destructive actions such as deletions, while the dialog-modal is a more generic modal window that may be used at any time.

To illustrate the use of modals, consider the following modal that confirms a user would like to delete their account:

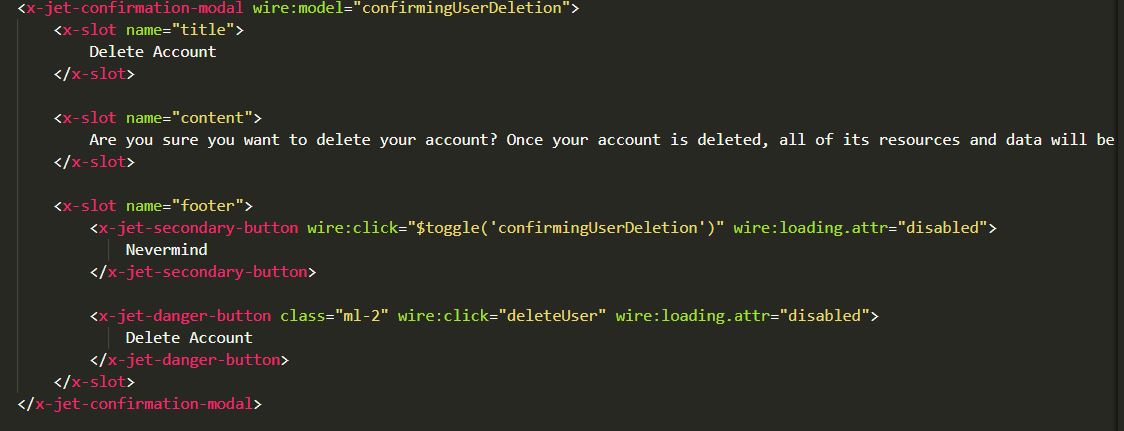


Image 30: Laravel Livewire -Models

As you can see, the modal's open / close state is determined by a wire:model property that is declared on the component. The property's name should correspond to a boolean property on your Livewire component's corresponding PHP class. Typically, you will set this property to true when the user clicks a UI element in your application that should open the modal. Of course, the property should be set to false when you are ready to close the modal.

The modal's contents may be specified by hydrating three Blade component slots: title, content, and footer.

References

1. <https://laravel.com/docs/9.x>
2. <https://www.javatpoint.com/laravel-sessions>