

Laravel

လို - တို - ရှင်း

အိမောင်

Fairway

© Copyright 2020, **Ei Maung**

Fairway Technology. All right reserved.

မာတိကာ

- 3 မိတ်ဆက်
- 4 အခန်း (၁) – PHP OOP
- 12 အခန်း (၂) – Namespaces
- 18 အခန်း (၃) – Laravel Project
- 23 အခန်း (၄) – Routing
- 30 အခန်း (၅) – Model – View – Controller
- 33 အခန်း (၆) – Controller
- 38 အခန်း (၇) – View
- 44 အခန်း (၈) – Migration and Model
- 54 အခန်း (၉) – Authentication
- 57 အခန်း (၁၀) – Master Template
- 65 အခန်း (၁၁) – Form
- 75 အခန်း (၁၂) – Relationship
- 85 အခန်း (၁၃) – Authorization
- 95 အခန်း (၁၄) – Basic API
- 103 အခန်း (၁၅) – Deployment
- 106 နိဂုံးချုပ်

မိတ်ဆက်

Laravel ဟာ လူကြိုက်များ ထင်ရှားနေတဲ့ PHP Framework တစ်ခုပါ။ Laravel ရဲ့ ကျစ်လစ်ရှင်းလင်းတဲ့ ဖွဲ့စည်းပုံကြောင့် ဒီ Framework ကိုသုံးပြီး ကုန်တွေရေးရတာ နှစ်လိုပျော်ရွှင်ဖွယ် ကောင်းပါတယ်။ အရင်က PHP ကုန်လို့ ပြောလိုက်ရင် ရှုပ်ယှက်ခတ်ပြီး ဖတ်ရခက်တဲ့ ကုန်တွေကို ပြေးမြင်ကြပေမယ့်၊ အခုနောက်ပိုင်းမှာတော့ PHP Language ကိုယ်တိုင်ရဲ့ တိုးတက်မှုတွေနဲ့အတူ Laravel ကို အသုံးများလာမှုကြောင့် PHP ကုန်ဆိုတာ သပ်သပ်ရပ်ရပ်နဲ့ ဖတ်ရှုနားလည်ရ လွယ်ကူတဲ့ကုန်တွေ ဖြစ်နေပါပြီ။

Laravel ရဲ့ အဓိကအားသာချက်ကတော့ လေ့လာရလွယ်ကူခြင်း ဖြစ်ပါတယ်။ ပြည့်စုံရှင်းလင်းတဲ့ Documentation ရှိတယ်။ အခြေခံကနေ အဆင့်မြင့်ထိ ဗွီဒီယိုသင်ခန်းစာတွေ စုံတယ်။ ဒါတင်မက မြန်မာဘာသာနဲ့ စာအုပ်တွေလည်း ရှိနေပါတယ်။ သင်တန်းတွေလည်း အများအပြား ရှိနေပါတယ်။ ဒီစာအုပ်ကလည်း အဲ့ဒီလို ပုံစံစုံရှိနေတဲ့ လေ့လာစရာများထဲက စာဖတ်သူများအတွက် အသုံးဝင်တဲ့ နောက်ထပ် လေ့လာစရာတစ်ခု ဖြစ်လိမ့်မယ်လို့ မျှော်လင့်ပါတယ်။

ဒီစာအုပ်ကို ဖတ်တဲ့သူဟာ PHP အခြေခံရှိထားပြီး ဖြစ်ဖို့လိုပါတယ်။ ဒီစာအုပ်ထဲက ကုန်နမူနာတွေကို လိုက်လုပ်နိုင်ဖို့အတွက် စာဖတ်သူရဲ့ စက်ထဲမှာ PHP ကုန်တွေ ရေးလို့/စမ်းလို့ ရတိုင်တဲ့ Development Environment တစ်ခု အသင့်ရှိနေပြီးလည်း ဖြစ်ဖို့လိုပါတယ်။ အသင့်မရှိသေးရင် အောက်မှာဖော်ပြထားတဲ့ Link ကနေ PHP Development Environment တည်ဆောက်ပုံ ဗွီဒီယို သင်ခန်းစာကို မဖြစ်မနေကြည့်ပြီး လိုက်လုပ်ထားဖို့ လိုအပ်ပါလိမ့်မယ်။

- <https://www.facebook.com/watch/?v=2994956967228569>

အခန်း (၁) - PHP OOP

Laravel ကို ကျင်ကျင်လည်လည် အသုံးပြုနိုင်ဖို့အတွက် PHP OOP ရေးထုံးတွေကို ကျေညက်ထားဖို့ လိုပါတယ်။ ဒါကြောင့် ပထမဦးဆုံးအနေနဲ့ PHP OOP ရေးထုံးကို ပြန်နွှေးပေးကြပါမယ်။ အစအဆုံးပြန်ပြောမှာတော့ မဟုတ်ပါဘူး။ လေ့လာဖူးပြီးသားလို့ သဘောထားပြီး မဖြစ်မနေ ပြန်ကြည့်သင့်တာလေးတွေကို ပြောသွားမှာပါ။ ပေးထားတဲ့ ကုဒ်နမူနာကို လေ့လာကြည့်ပါ။

```
<?php

class Animal
{
    public $name;
}

$dog = new Animal;
$dog->name = "Bobby";
echo $dog->name;           // Bobby

$dog->legs = 4;
echo $dog->legs;           // 4
```

ဒါဟာ သတိပြုစရာအချို့ပါတဲ့ အခြေခံအကျဆုံး PHP OOP ကုဒ်ပါ။ ပထမဆုံး Naming Convention ကနေစ စပြောပါမယ်။ Programming မှာ အမည်တွဲပေးကြတဲ့အခါ snake_case, camelCase နဲ့ CapitalCase ဆိုပြီး နည်းလမ်း (၃) မျိုးကို သုံးကြလေ့ရှိပါတယ်။ Class Name တွေ ပေးတဲ့အခါ CapitalCase နဲ့ပေးရပါတယ်။ ဥပမာ - MathHelper, PostController, ViewModel စသဖြင့်။ တစ်ခြားရေးနည်းတွေ သုံးရင်လည်းရပေမယ့် အများကစံထားပြီး CapitalCase ကို သုံးတဲ့

အတွက် ကျွန်တော်တို့ကလည်း Class Name တွေ ပေးတိုင်းမှာ CapitalCase ကိုပဲ သုံးပြီး ပေးကြရမှာ ဖြစ်ပါတယ်။

Property တွေ Method တွေကိုတော့ camelCase နဲ့ ပေးရပါတယ်။ ဥပမာ - firstName, sayHello, saveUser စသဖြင့်။ camelCase က CapitalCase နဲ့ အတူတူပါပဲ။ ရှေ့ဆုံးစာလုံး ကို အသေးနဲ့ရေးတာ တစ်ခုပဲ ကွာသွားတာပါ။

Variable Name တွေအတွက်တော့ snake_case ကို သုံးပါတယ်။ Database Name တွေ၊ Table Name တွေနဲ့ Column Name တွေကိုလည်း snake_case ပဲ သုံးကြပါတယ်။ ဥပမာ - category_id, created_on, updated_on စသဖြင့်။

ဒါ အရေးကြီးပါတယ်။ Laravel က Convention Over Configuration လို့ခေါ်တဲ့ နည်းစနစ်တစ်မျိုးကို သုံးပါတယ်။ လိုရင်းက၊ အမည်ကိုမှန်အောင်ပေးရင် တစ်ချို့အလုပ်တွေကို သူ့အလိုအလျောက် အလုပ်လုပ် ပေးနိုင်တဲ့စနစ် မျိုးပါ။ ဒါကြောင့် အမည်ပေးတာသာ မှန်မယ်ဆိုရင် အကုန်လုံးကို ကိုယ်တိုင်လိုက် ရေးနေ စရာ မလိုဘဲ Framework အလိုအလျောက် လုပ်ပေးသွားနိုင်တာတွေ ရှိပါတယ်။ အမည်ပေးပုံ Naming Convention နဲ့ပတ်သက်ပြီး အခုလို အနှစ်ချုပ် မှတ်ထားပါ။

1. Class Name တွေအတွက် CapitalCase ကိုသုံး
2. Method Name တွေ Property Name တွေအတွက် camelCase ကိုသုံး
3. Variable တွေ Database, Table, Column Name တွေအတွက် snake_case ကိုသုံး

ဆက်လက်ပြီး၊ အပေါ်က ပေးထားတဲ့ ကုဒ်နမူနာကို ပြန်ကြည့်ပါ။ Animal Class မှာ \$name Property ပါဝင်ပါတယ်။ public Modifier ကို အသုံးပြု ကြေညာထားပါတယ်။ Animal Class ကို အသုံးပြုပြီး \$dog Object ကို တည်ဆောက်တဲ့အခါ \$dog Object မှာ \$name Property ပါဝင်သွားပါတယ်။ နမူနာ အနေနဲ့ \$name Property အတွက်တန်ဖိုးသတ်မှတ်ပြီး ပြန်လည်အသုံးပြုပြ ထားပါတယ်။ ထူးခြားချက် အနေနဲ့ PHP မှာ Object ရဲ့ Property တွေ Method တွေကို Access လုပ်ဖို့အတွက် Operator အဖြစ် Dot ကို မသုံးပဲ Dart (မျှားသင်္ကေတ ->) ကို အသုံးပြုခြင်းပဲဖြစ်ပါတယ်။ ဥပမာ - တစ်ခြား Language အများ စုမှာ dog.name လို့ ရေးနိုင်ပေမယ့် PHP မှာတော့ \$dog->name လို့ ရေးရပါတယ်။ နောက်ထပ်

ထူးခြားချက်ကတော့ မူလ Object တည်ဆောက်စဉ်မှာ မပါဝင်တဲ့ Property ကိုလည်း တိုက်ရိုက် အသုံးပြု နိုင်ကြောင်း \$legs Property နဲ့ နမူနာပြပေးထားပါတယ်။ နည်းနည်းပိုပြည့်စုံသွားအောင် ထပ်ဖြည့် ကြည့်ပါမယ်။

```
<?php

class Animal
{
    private $name;

    public function __construct($name)
    {
        $this->name = $name;
    }

    public function sayName()
    {
        return "Hello, my name is $this->name";
    }

    static function info()
    {
        return "Animal class";
    }
}

$dog = new Animal("Bobby");
echo $dog->sayName(); // Hello, my name is Bobby
```

ကျန်တာတွေ မပြောခင် ဟိုးထိပ်ဆုံးက <?php ကုဒ်အဖွင့်သင်္ကေတအကြောင်း အရင်ပြောပါမယ်။ ပုံမှန် အားဖြင့် PHP ကုဒ်တွေကို <?php အဖွင့်နဲ့ ?> အပိတ်ကြားထဲမှာ ရေးရလေ့ရှိပါတယ်။ ဒါပေမယ့် လက်ရှိ ကုဒ်ဖိုင်ထဲမှာ PHP ကုဒ်တွေချည်းပဲ ပါတယ်ဆိုရင် (HTML တွေဘာတွေနဲ့ ရောမရေးဘူးဆိုရင်) အပိတ် ?> က ထည့်စရာမလိုပါဘူး။ မထည့်တာပိုကောင်းပါတယ်။ မထည့်တာ ပိုကောင်းတယ်လို့ပဲ မှတ်ထားပေးပါ။ အကြောင်းရင်းကိုတော့ ပေရှည်မှာစိုးလို့ ထည့်မပြောတော့ပါဘူး။

ရေးထားတဲ့ကုဒ်ကိုလေ့လာကြည့်ရင် \$name Property က private ဖြစ်သွားပါပြီ။ ဒါကြောင့် ဒီ Class ထဲမှာပဲ သုံးလို့ရတော့မှာပါ။ အပြင်ကနေ ယူသုံးလို့ မရတော့ပါဘူး။ ထူးခြားချက်အနေနဲ့ Constructor ပါသွားပါပြီ။ ရှေ့ဆုံးက Underscore (_) ခုနဲ့စတဲ့ __construct() ဆိုတဲ့ Method ဟာ PHP Class

တွေအတွက် Constructor ဖြစ်ပါတယ်။ ဒါကြောင့် Object တည်ဆောက်တဲ့အခါတိုင်း ဒီ Method အလုပ် လုပ်သွားမှာ ဖြစ်ပါတယ်။ Constructor Method က \$name Parameter ထည့်သွင်း ကြေညာထားလို့ Object တည်ဆောက်ရင် \$name ပေးပြီးတော့မှပဲ ဆောက်လို့ရတော့မှာပါ။

ဆက်လက်ပြီး Property တွေ Method တွေကို Class အတွင်းထဲမှာ ယူသုံးဖို့အတွက် \$this ကနေ တစ်ဆင့် အသုံးပြုရတာကိုလည်း သတိပြုပါ။ ကျန်တာတွေကိုတော့ စာနဲ့ရေးပြီး ရှင်းနေရင် ပိုရှုပ်ပါတယ်။ အပေါ်ကနမူနာမှာ ရေးထားတဲ့ကုဒ်ကို တိုက်ရိုက် ဖတ်ပြီးတော့သာ လေ့လာကြည့်လိုက်ပါ။

နမူနာမှာ static Method တစ်ခုလည်း ပါဝင်ပါတယ်။ static Property တွေ static Method တွေကိုတော့ Object ကနေတစ်ဆင့် သွားစရာမလိုဘဲ Class Name ကနေ တိုက်ရိုက်ခေါ်ယူ အသုံးပြုလို့ ရပါတယ်။ ဒီလိုအသုံးပြုနိုင်ဖို့အတွက် Scope Resolution Operator (: :) ကို သုံးရပါတယ်။ ဥပမာ -

```
echo Animal::info(); // Animal class
```

Property တွေ Method တွေ ကြေညာတဲ့အခါ public, private, protected, static စ သဖြင့် Access Control Modifier တွေ မပါဘဲလည်း ကြေညာလို့ရပါတယ်။ Access Control Modifier တွေ မပါရင်တော့ Default က public ဖြစ်ပါတယ်။ Inheritance ရေးထုံးကို ဆက်လက်ဖော်ပြပါမယ်။

```
class Dog extends Animal
{
    //
}

$bobby = new Dog("Bobby");
echo $bobby->sayName(); // Hello, my name is Bobby
```

ဒီနေရာမှာတော့ မြင်သာတဲ့ထူးခြားချက် မရှိပါဘူး။ OOP Language အများစု နည်းတူ extends ကို အသုံးပြုပြီး Inherit လုပ်လို့ရပါတယ်။ နမူနာမှာ Dog Class ဟာ Animal ကို Inherit လုပ်တဲ့အတွက် Animal Class ရဲ့ လုပ်ဆောင်ချက်တွေကို ဆက်ခံရရှိနေတယ်ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ PHP မှာ Multiple Inheritance မရတာကိုတော့ ထည့်သွင်းမှတ်သားသင့်ပါတယ်။

Interfaces

Laravel ကုဒ်တွေကို နားလည်ဖို့အတွက် Interface ရဲ့ သဘောသဘာဝကို သိရှိထားဖို့ဟာလည်း မဖြစ်မနေ လိုအပ်ချက်ဖြစ်ပါတယ်။ ဒီကုဒ်ကို လေ့လာကြည့်ပါ။

```
class Fox
{
    public function run()
    {
        return "The fox is running";
    }
}

class Wolf
{
    public function run()
    {
        return "The wolf is running";
    }
}

function app(Fox $fox)
{
    echo $fox->run();
}

app(new Fox);           // The fox is running
app(new Wolf);          // Error: must be Fox, Wolf given
```

နမူနာမှာ Fox နဲ့ Wolf ဆိုပြီး Class နှစ်ခုရှိပါတယ်။ app() Function က Fox Object ကို Parameter အနေနဲ့ လက်ခံအလုပ်လုပ်ပါတယ်။ ဒါကြောင့် Fox Object ကို ပေးပြီးခေါ်တဲ့အခါ အလုပ်လုပ်ပေးမယ့်၊ Wolf Object ကိုပေးပြီး ခေါ်တဲ့အခါ အလုပ်မလုပ်တော့တာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ သေချာကြည့်ရင် Fox နဲ့ Wolf ဟာ အလုပ်လုပ်ပုံ အတူတူပါပဲ။ ဒီလိုမျိုး အမျိုးအစားမတူပေမယ့် အလုပ်လုပ်ပုံတူတဲ့ Object တွေကို ဖလှယ်အသုံးပြုနိုင်ဖို့ Interface ကို အသုံးပြုခြင်း ဖြစ်ပါတယ်။ ဒီကုဒ်ကို Interface တစ်ခုခံပြီး အခုလို ပြင်ရေးကြည့်ပါ။

```

interface Dog
{
    public function run();
}

class Fox implements Dog
{
    public function run()
    {
        return "The fox is running";
    }
}

class Wolf implements Dog
{
    public function run()
    {
        return "The wolf is running";
    }
}

function app(Dog $dog)
{
    echo $dog->run();
}

app(new Fox);           // The fox is running
app(new Wolf);          // The wolf is running

```

နမူနာမှာ Dog Interface ပါဝင်သွားပြီး Fox နဲ့ Wolf နှစ်ခုစလုံးက Dog ကို Implement လုပ်ထားကြပါတယ်။ ဒါကြောင့် Fox နဲ့ Wolf ဟာ အမျိုးအစားမတူပေမယ့် မူလဇစ်မြစ်တူသွားကြပါပြီ။ Interface ဆိုတာ တစ်ကယ်အလုပ်လုပ်တဲ့ ကုဒ်တွေ မပါပါဘူး။ ကြေညာချက်တွေပဲ ပါပါတယ်။ ဒါကြောင့် Dog Interface မှာ public function run() ကို ကြေညာယုံပဲ ကြေညာထားပါတယ်။ ဒီလိုကြေညာထားတဲ့ အတွက် သူကို Implement လုပ်ထားကြတဲ့ Fox နဲ့ Wolf တို့က run() အတွက် ကုဒ်ကို မဖြစ်မနေရေးပေးရမှာ ဖြစ်ပါတယ်။ မရေးရင် Error တက်ပါတယ်။

ဆက်လေ့လာကြည့်လိုက်ရင် app() Function က Dog Object ကို လက်ခံတာ ဖြစ်သွားပါပြီ။ Dog Interface ကနေတော့ Object ဆောက်လို့ မရပါဘူး။ ဒါပေမယ့်သူ့ကို Implement လုပ်ထားတဲ့ Fox တို့ Wolf တို့ကနေ Object ဆောက်ပြီး အသုံးပြုတဲ့အခါ Object အမျိုးအစား မတူပေမယ့် အခြေခံတူလို့ အလုပ်လုပ်တယ်ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

Interface နဲ့ သဘောသဘာဝ ဆင်တာကတော့ Abstract Class ဖြစ်ပါတယ်။ ကွာသွားတာက Interface မှာ ကြေညာချက်တွေပဲ ရှိရပြီး Abstract Class ကတော့ ကြေညာချက်ရော အလုပ်လုပ်တဲ့ ကုဒ်ပါ နှစ်မျိုးလုံး ပါလို့ရတာပါ။ နမူနာထည့်မပြတော့ပါဘူး။ လို - တို - ရှင်း ဆိုတဲ့အတိုင်း မဖြစ်မနေလိုတာကိုပဲ ရွေးပြီး ပြောချင်လို့ပါ။ အထက်က Interface ကုဒ်ကို နားလည်မယ်ဆိုရင် Laravel ကို ရှေ့ဆက်လေ့လာတဲ့အခါ ကိုယ့်အတွက် အများကြီး အထောက်အကူဖြစ်စေမယ်ပါလိမ့်မယ်။

Traits

PHP OOP မှာ Trait လို့ခေါ်တဲ့ ရေးထုံးတစ်မျိုးလည်း ရှိပါသေးတယ်။ ဒါလည်းပဲ Laravel ကုဒ်တွေကို နားလည်ဖို့အတွက် သိရှိထားဖို့ လိုအပ်တဲ့ ရေထုံးဖြစ်ပါတယ်။ လိုရင်းကတော့ တစ်ချို့ ထပ်ခါထပ်ခါ ပြန် ရေးရလေ့ရှိတဲ့ ကုဒ်တွေကို ရေးစရာမလိုတော့ပဲ ခေါ်သုံးနိုင်စေတဲ့ ရေးနည်းပါ။ ဥပမာ -

```
trait BasicCalculation
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}

class Math
{
    use BasicCalculation;

    public $PI = 3.14;

    public function circleArea($r)
    {
        return $this->PI * $r * $r;
    }
}

$math = new Math;
echo $math->add(1, 2);           // 3
```

နမူနာမှာ BasicCalculation အမည်နဲ့ Trait တစ်ခုကြေညာထားပြီး သူ့ထဲမှာ add() Function ပါပါတယ်။ Math Class က use BasicCalculation ဆိုပြီး အဲ့ဒီ Trait ကို ခေါ်သုံးလိုက်တဲ့အခါ BasicCalculation မှာ ရေးထားတဲ့ လုပ်ဆောင်ချက်ကို Math Class က ရရှိသွားခြင်းပဲ ဖြစ်ပါတယ်။

OOP အကြောင်းပြောတဲ့အခါ တစ်ကယ်တော့ အခုဖော်ပြခဲ့တဲ့ ရေးထုံးလို ကိစ္စမျိုးတွေက လွယ်ပါတယ်။ တစ်ကယ့် အနှစ်သာရကို လေ့လာချင်ရင် SOLID လို Design Principle တွေ Factory Pattern, Singleton Pattern, Adapter Pattern, Facade Pattern စသဖြင့် Design Pattern တွေကို ဆက်လေ့လာရမှာပါ။ ကျယ်ပြန့်ပြီး နက်နဲတဲ့ အကြောင်းအရာတွေ ဖြစ်ပါတယ်။ ဒါတွေသိထားရင် Laravel ကို လေ့လာတဲ့နေရာ မှာသာမက ကိုယ့်ရဲ့ ကုန်ရေးသားမှုကဏ္ဍအားလုံးမှာ ပြုပြင်ထိမ်းသိမ်းရလွယ်ကူပြီး အဆင့်မြင့်တဲ့ ကုန်တွေ ကို ရေးသားနိုင်မှာပါ။ ဒါပေမယ့် ဒါတွေကို မဖြစ်မနေသိထားမှ Laravel ကို ဆက်လုပ်လို့ရတာမျိုးတော့ မဟုတ်ပါဘူး။ ဒါကြောင့် ဒီစာအုပ်မှာတော့ အဲ့ဒီအကြောင်းတွေ အကျယ်မချဲ့ပါဘူး။ ဆက်လက်လေ့လာလို ရင် အောက်မှာပေးထားတဲ့ လိပ်စာက ကုန်နမူနာတွေကနေတစ်ဆင့် လေ့လာလို့ ရပါတယ်။

- <https://github.com/eimg/design-patterns-php>

အခန်း (၂) - Namespaces

PHP မှာ Function Name တွေ Class Name တွေ တူရင် လက်မခံပါဘူး။ ဒါကြောင့် ဒီကုဒ်ကို စမ်းကြည့်ရင် Error ဖြစ်ပါလိမ့်မယ်။

```
// math.php
function add($a, $b) {
    return $a + $b;
}
```

```
// calc.php
function add($nums) {
    return array_sum($nums);
}
```

```
// app.php
include("math.php");
include("calc.php");           // Error: Cannot redeclare add

echo add(1, 2);                // Which add()? math.php or calc.php?
```

နမူနာအရ math.php ထဲမှာ add() Function ရှိနေပြီး၊ calc.php ထဲမှာလည်း add() အမည်နဲ့ Function ရှိနေပါတယ်။ ဒါကြောင့် ဒီဖိုင်နှစ်ခုလုံးကို Include လုပ်ပြီး သုံးဖို့ကြိုးစားတဲ့အခါ Function အမည်တူ နှစ်ခု တိုက်နေပြီး Error တက်ပါလိမ့်မယ်။ ဒီပြဿနာကို Namespace ရေးထုံးနဲ့ ဖြေရှင်းလို့ရပါတယ်။ ဒီလိုပါ -

```
// math.php
namespace Math;

function add($a, $b) {
    return $a + $b;
}
```

```
// calc.php
namespace Calc;

function add($nums) {
    return array_sum($nums);
}
```

```
// app.php
include("math.php");
include("calc.php");

echo Math\add(1, 2);           // 3
```

namespace ရေထုံးအမျိုးမျိုးရှိပါတယ်။ ဒါပေမယ့် အခုနမူနာပေးထားတဲ့ တစ်မျိုးကြည့်ထားရင် ရပါပြီ။ ကုဒ်ဖိုင်တစ်ခုတိုင်းမှာ သူ့ကိုယ်ပိုင် Namespace တစ်ခု ပေးထားတဲ့သဘောပါ။ တစ်ခြားကုဒ်တွေ မလာခင် ဖိုင်ရဲ့ အပေါ်ဆုံးမှာ ရေးရပါတယ်။ Namespace အမည်ကိုလည်း CapitalCase နဲ့ပဲ ပေးသင့်ပါတယ်။

နောက်ဆုံး app.php မှာ ပြန်ခေါ်သုံးပုံကို သတိပြုပါ။ Namespace အပြည့်အစုံနဲ့ ပြန်ခေါ်သုံးပေးရပါတယ်။ Backslash (\) ကို သုံးရပါတယ်။ နမူနာအရ add() Function ကို ခေါ်တဲ့အခါ Math\add() လို့ ခေါ်ထားတဲ့အတွက် add() Function နှစ်ခုရှိပေမယ့် Math Namespace အောက်က add() အလုပ်လုပ် သွားမှာပါ။ ဒီလိုလည်း ဖြစ်နိုင်ပါသေးတယ်။

```
// app.php
namespace Math;

include("math.php");
include("calc.php");

echo add(1, 2);           // 3
```

ဒီတစ်ခါတော့ `app.php` ကိုယ်တိုင်မှာ `Namespace` ရှိသွားပါပြီ။ `Namespace` က `Math` ဖြစ်တဲ့အတွက် `add()` ကို ခေါ်တဲ့အခါ `Namespace` မထည့်တော့ဘဲ ခေါ်ထားပါတယ်။ `Namespace` တူရင် ထည့်စရာ မလိုတဲ့အတွက်ပါ။ `Calc` `Namespace` အောက်က `add()` ကို လှမ်းခေါ်ချင်ရင်တော့ အခုလို ခေါ်နိုင်ပါတယ်။

```
echo \Calc\add([1, 2]);           // 3
```

ဟိုးရှေ့ဆုံးက `Backslash` ကို သတိပြုပါ။ အဲ့ဒါ `Root Namespace` ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ အဲ့ဒီ `Backslash` မပါဘဲ `Calc\add()` လို့ ခေါ်မယ်ဆိုရင် `Namespace` လမ်းကြောင်း အပြည့်အစုံက `Math\Calc\add()` ဖြစ်သွားမှာပါ။ သူ့ကိုယ်တိုင်က `Math` `Namespace` အောက် ရောက်နေလို့ပါ။ `Math\Calc\add()` ဆိုတာ မရှိတဲ့ အတွက် `Error` တက်ပါလိမ့်မယ်၊ ဒါကြောင့် ဟိုးရှေ့ဆုံးက `Backslash` နဲ့ `Root Name` အပါအဝင် အပြည့်အစုံ သုံးပြီး ခေါ်ရတာပါ။

`Namespace` တွေကို အဆင့်ဆင့်လည်း ပေးလို့ရပါသေးတယ်။ ဒီလိုပါ -

```
// math.php
namespace Libs\Math;

function add($a, $b) {
    return $a + $b;
}
```

```
// calc.php
namespace Helper\Calc;

function add($nums) {
    return array_sum($nums);
}
```

```
// app.php
namespace Libs;

include("math.php");
include("calc.php");

echo Math\add(1, 2);           // 3
```

math.php ရဲ့ Namespace က Libs\Math ဖြစ်သွားပါပြီ။ app.php ရဲ့ Namespace ကတော့ Libs ဖြစ်တဲ့အတွက် add() Function ကိုခေါ်တဲ့အခါ Math\add() လို့ခေါ်လိုက်ရင် ရသွားပါပြီ။ သူ့ကိုယ်ပိုင် Namespace နဲ့ ပေါင်းလိုက်ရင် အပြည့်အစုံက Libs\Math\add() ဖြစ်သွားမှာမို့လို့ပါ။

ဒါဟာ အလွန်အသုံးဝင်ပြီး Laravel မှာလည်း ကျယ်ကျယ်ပြန့်ပြန့် အသုံးပြုမယ့် ရေးထုံးဖြစ်ပါတယ်။ လက်တွေ့မှာ Namespace လမ်းကြောင်းကို ဖိဒါ/ဖိုင် Path လမ်းကြောင်းအတိုင်း ပေးကြလေ့ ရှိပါတယ်။ ဒီအတွက် PSR-4 လို့ခေါ်တဲ့ Standard တစ်ခုရှိပါတယ်။ Framework တီထွင်သူတွေဆိုရင် မဖြစ်မနေ လေ့လာကြရမှာပါ။ ကျွန်တော်တို့ကတော့ Framework တီထွင်ချင်တာ မဟုတ်ပါဘူး။ ရှိနေတဲ့ Framework ကို အသုံးပြုချင်တာပါ။ ဒါကြောင့် အဲဒီကိစ္စကို အသေးစိတ် ထည့်မကြည့်ပါဘူး။

ဥပမာအားဖြင့် ဖိဒါ/ဖိုင် Path လမ်းကြောင်းက libs/helper/math/add.php ဆိုရင် အဲဒီဖိုင်ရဲ့ Namespace က အခုလို ဖြစ်သင့်ပါတယ်။

```
namespace Libs\Helper\Math;
```


ဖိုဒါ/ဖိုင် အမည်ကို စာလုံးအသေး/အကြီး ဘယ်လိုပေးထားပါစေ Namespace ကတော့ CapitalCase ကို ပဲ သုံးတယ်ဆိုတာ သတိပြုပါ။ ဖိုင်အမည်ကို Namespace မှာ ထည့်ပေးကြလေ့ မရှိပါဘူး။ ဒါလေးကိုသေချာမှတ်ထားပေးပါ။

ဆက်လက်ပြီး Class တွေနဲ့ တွဲဖက်အသုံးပြုပုံကို ဆက်ကြည့်ကြပါမယ်။ ဒီနေရာမှာလည်း အမည်ပေးပုံက စကားပြောပါတယ်။ ဖိုင်အမည်နဲ့ Class အမည်ကို တူအောင်ပေးသင့်ပါတယ်။ Convention Over Configuration အကြောင်းပြီးခဲ့တဲ့ အခန်းမှာ ပြောခဲ့ပါတယ်။ အမည်မှန်အောင် ပေးယုံနဲ့ အဆင်ပြေသွားမှာတွေ အများကြီး ရှိနေလို့ အရေးကြီးပါတယ်။ ကောင်းကောင်းသတိပြုပါ။ Laravel မှာ Class Name ကို CapitalCase ပေးသလို ဖိုင် Name ကိုလည်း CapitalCase ပေးလေ့ရှိပါတယ်။ မျက်စိမလည်ပါစေနဲ့၊ ဒီလောက်ကြီး မခက်လှပါဘူး။ သိပ်မရှင်းရင် နောက်တစ်ခေါက်လောက် ပြည့်ကြည့်ထားပေးပါ။

```
// libs\helper\Math.php

namespace Libs\Helper;

class Math
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

```
// index.php

include("libs\helper\Math.php");

$math = new Libs\Helper\Math;
echo $math->add(1, 2); // 3
```

နမူနာမှာ Math Class ကို အသုံးပြုလိုတဲ့အခါသူရဲ့ Namespace လမ်းကြောင်း အပြည့်အစုံ သုံးရတာကို တွေ့ရပါလိမ့်မယ်။ အဲဒီလို လမ်းကြောင်းအပြည့်အစုံ ပေးစရာမလိုအောင် Import လုပ်လို့ရပါတယ်။ ရေးပုံ ရေးနည်းက ဒီလိုပါ။

```
// index.php

include("libs\helper\Math.php");

use Libs\Helper\Math;

$math = new Math;
echo $math->add(1, 2);           // 3
```

use ကိုအသုံးပြုပြီး Namespace ကို Import လုပ်ထားပြီးဖြစ်လို့ အသုံးပြုတဲ့အခါ Namespace လမ်းကြောင်း အပြည့်အစုံပေးစရာမလိုတော့ပါဘူး။ Class Name နဲ့ တိုက်ရိုက်သုံးလို့ ရသွားပါပြီ။

နောက်တစ်ဆင့်အနေနဲ့ ဖိုင်တွေကို include() လုပ်စရာမလိုဘဲ သုံးလို့ရအောင် လုပ်တဲ့နည်းကို လေ့လာရမှာပါ။ Class Autoload လို့ခေါ်ပါတယ်။ လိုရင်းကတော့ Class တစ်ခုကို အသုံးပြုလိုက်တာနဲ့ အဲ့ဒီ Class ရှိနေတဲ့ဖိုင်ကို အလိုအလျောက် include() လုပ်ပေးစေတဲ့နည်းဖြစ်ပါတယ်။ ဒီနည်းနဲ့ဆိုရင် ကိုယ်တိုင် include() လုပ်ပေးစရာမလိုတော့ပါဘူး။ ဒီစာအုပ်မှာ Class Autoload အကြောင်းကို ထည့်ရှင်းမပြတော့ပါဘူး။ အခုဖော်ပြထားတဲ့ Namespace ရေးထုံးကို ကောင်းကောင်းနားလည်မယ်ဆိုရင် Laravel ကုဒ်တွေကိုလေ့လာတဲ့ အခါမှာ အထောက်အကူဖြစ်နေပါပြီ။

Class Autoload အကြောင်းကို အောက်မှာပေးထားတဲ့ လိပ်စာကနေ ဗွီဒီယိုသင်ခန်းစာအနေနဲ့ လေ့လာနိုင်ပါတယ်။

- <https://www.facebook.com/watch/?v=1598184780191951>

အခန်း (၃) - Laravel Project

Wordpress လို နည်းပညာမျိုးအပါအဝင် အရင်တုန်းက PHP နည်းပညာတွေကို လိုချင်ရင် သက်ဆိုင်ရာဝတ်ဆိုင်ကနေ တိုက်ရိုက် Download ရယူပြီး သုံးရပါတယ်။ ကနေ့ခေတ် PHP နည်းပညာတွေကတော့ အဲဒီလို တိုက်ရိုက် Download လုပ်တဲ့နည်းလမ်းကို မသုံးတော့ဘဲ Composer လို့ခေါ်တဲ့ Dependency Manager နည်းပညာကိုသုံးကြပါတယ်။ Package Manager လို့လည်းခေါ်ကြပါတယ်။ တစ်ခြား Language တွေမှာလည်း အလားတူ နည်းပညာတွေ ရှိပါတယ်။ ဥပမာ - JavaScript အတွက်ဆိုရင် NPM, Yarn စတဲ့ Package Manager နည်းပညာတွေ ရှိပါတယ်။

ဒီနည်းပညာတွေရဲ့ လိုရင်းအနှစ်ချုပ်ကတော့ အသုံးပြုလိုတဲ့ Package တစ်ခုကို ရယူလိုက်တဲ့အခါ သူနဲ့ ဆက်စပ်လိုအပ်တဲ့ Package တွေကို အလိုအလျောက် ရယူပေးနိုင်ခြင်း ဖြစ်ပါတယ်။ ဥပမာ - Laravel အလုပ်လုပ်ဖို့အတွက် Symfony Http Kernel လိုပါတယ်။ ဒါပေမယ့် Composer နဲ့ Laravel ကို ရယူလိုက်မယ်ဆိုရင် Composer က Symfony Http Kernel ကို အလိုအလျောက် တစ်ပါတည်း ရယူပေးသွားမှာ ဖြစ်ပါတယ်။ Laravel က ဒီလိုမျိုး ဆက်စပ်နည်းပညာပေါင်း များစွာကို အသုံးပြုထားပေမယ့် ကိုယ့်ဘာသာ တစ်ခုချင်း လိုက်ဒေါင်းစရာ မလိုတော့ဘဲ Composer က လိုသမျှအကုန် အလိုအလျောက် ယူပေးသွားတဲ့ သဘောပါ။

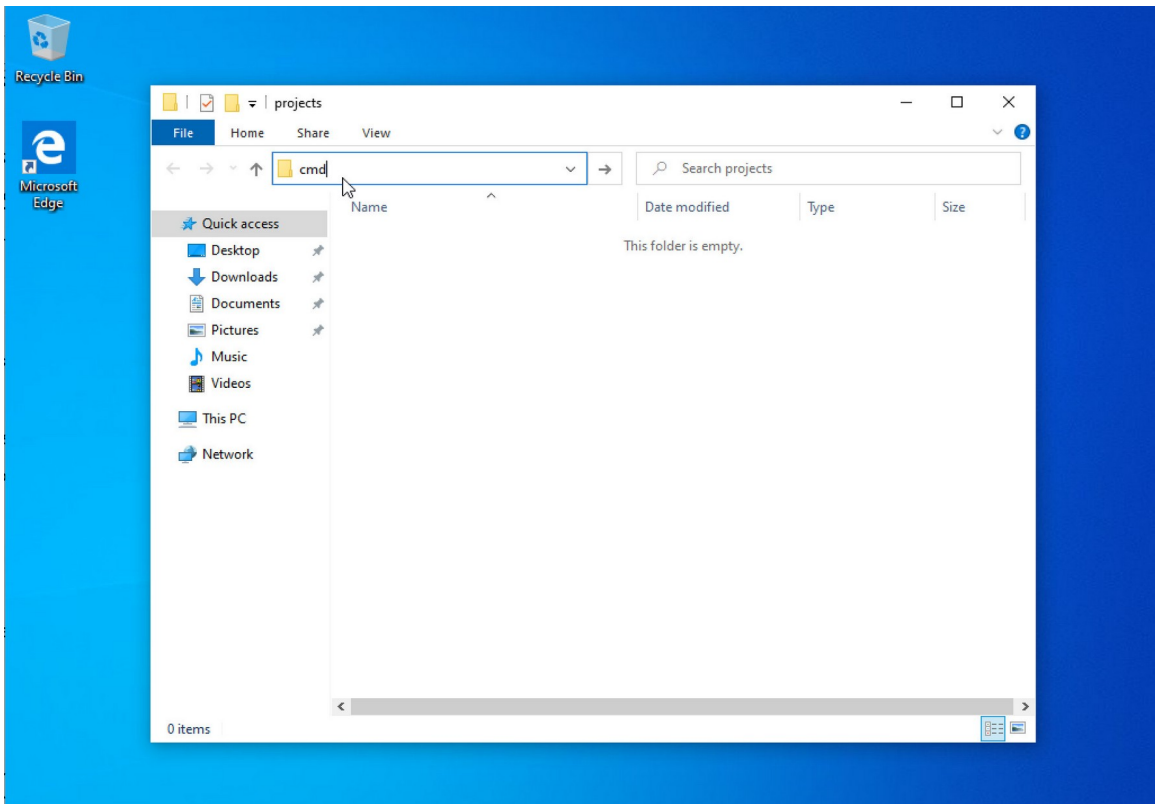
Composer ကို Install လုပ်ပုံလုပ်နည်းကို PHP Development Environment တည်ဆောက်ပုံ ဗွီဒီယို သင်ခန်းစာမှာ ထည့်ပြီး ရှင်းပြထားပါတယ်။ <https://getcomposer.org> ဝတ်ဆိုင်ကနေရယူနိုင်ပါတယ်။ Composer ဟာ Command Line နည်းပညာဖြစ်ပြီး ဒီနည်းပညာအကြောင်းလေ့လာချင်တယ်ဆိုရင် အောက်မှာပေးထားတဲ့ လိပ်စာကနေတစ်ဆင့် ဗွီဒီယိုသင်ခန်းစာအနေနဲ့ လေ့လာနိုင်ပါတယ်။

Composer - <https://www.facebook.com/watch/?v=1598556580154771>

Creating Laravel Project

Laravel ရဲ့ Documentation ဖြစ်တဲ့ laravel.com/docs မှာသွားပြီးလေ့လာကြည့်လိုက်ရင် Laravel ပရောဂျက် တည်ဆောက်ပုံ (၂) နည်း ပေးထားတာကို တွေ့ရနိုင်ပါတယ်။ ပထမနည်းက laravel Installer ကို Composer နဲ့ အရင် Install လုပ်ပြီးမှ ပရောဂျက်တည်ဆောက်တဲ့နည်းပါ။ ဒုတိယနည်းကတော့ Composer ကိုပဲ တိုက်ရိုက်အသုံးပြုပြီး ပရောဂျက်တည်ဆောက်တဲ့နည်းပါ။ အဲ့ဒီဒုတိယနည်းကိုပဲ ဒီနေရာမှာ ဖော်ပြပါမယ်။

ပထမဆုံးအနေနဲ့ ပရောဂျက်ဖိုဒါတည်ဆောက်လိုတဲ့နေရာမှာ Command Prompt (သို့) Terminal ကို ဖွင့်လိုက်ပါ။ **Tip** – Windows Explorer ရဲ့ Address Bar ထဲမှာ `cmd` [Enter] နှိပ်ခြင်းအားဖြင့် ရောက်ရှိနေတဲ့ ဖိုဒါထဲမှာ Command Prompt ကို ဖွင့်လို့ရပါတယ်။



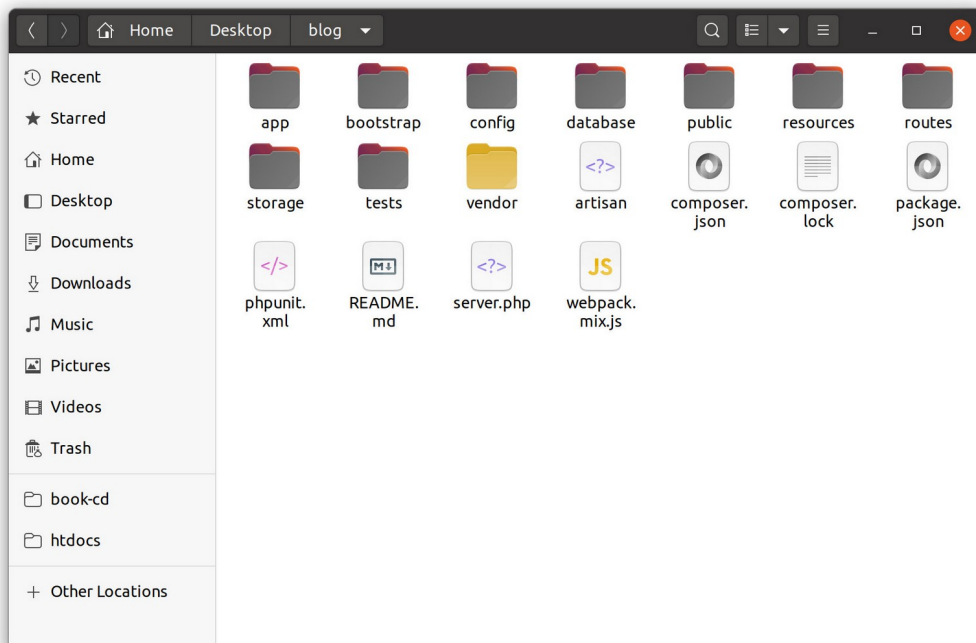
Windows မှာ Command Prompt လို့ခေါ်ပြီး တစ်ခြား OS တွေမှာ Terminal လို့ခေါ်ပါတယ်။ ရှေ့လျှောက် နှစ်မျိုး ပြောမနေတော့ပါဘူး။ Terminal လို့ပဲ သုံးနှုံးပြီး ဆက်ပြောသွားပါမယ်။ Terminal ကို ဖွင့်

ပြီးရင် ဒီ Command ကို Run ရမှာပါ။

```
composer create-project laravel/laravel blog
```

composer Command အတွက် create-project Option ကိုပေးလိုက်တာပါ။ ဒါကြောင့် Composer က ကျွန်တော်တို့အတွက် ပရောဂျက်ဖိုဒါတစ်ခု ဆောက်ပေးသွားပါလိမ့်မယ်။ နောက်ကနေ အသုံးပြုလိုတဲ့ Vendor/Package ကို ပေးရပါတယ်။ နမူနာအရ laravel အမည်ရ Vendor ကပေးထားတဲ့ laravel အမည်ရ Package ကို အသုံးပြုမယ်ဆိုတဲ့ အဓိပ္ပါယ်ဖြစ်ပါတယ်။ နောက်ဆုံးက blog ကတော့ အသုံးပြုလိုတဲ့ ဖိုဒါအမည် ဖြစ်ပါတယ်။ ကြိုက်တဲ့အမည် ပေးလို့ရပါတယ်။ လိုချင်တဲ့ Laravel Version နံပါတ်ကို ပြောလို့လည်း ရပါသေးတယ်။ နမူနာမှာ ပြောမထားတဲ့အတွက် ကိုယ် Install လုပ်ထားတဲ့ PHP Version နဲ့ ကိုက်ညီတဲ့ နောက်ဆုံး Laravel Version ကို ရယူပေးသွားမှာ ဖြစ်ပါတယ်။

Composer က laravel ကို Download ယူပေးပြီးတာနဲ့ တစ်ဆက်တည်း လိုအပ်တဲ့ ဆက်စပ် နည်းပညာတွေကိုပါ ဆက်တိုက် Download ယူပေးသွားမှာပါ။ ဒါကြောင့် နည်းနည်းတော့ အချိန်ပေးပြီး စောင့်ရနိုင်ပါတယ်။ ပြီးသွားတဲ့အခါ blog အမည်နဲ့ ပရောဂျက်ဖိုဒါကို ရမှာဖြစ်ပြီး၊ ပရောဂျက်ဖိုဒါထဲက vendors ထဲမှာ laravel နဲ့အတူ ဆက်စပ်လိုအပ်တဲ့ Package အားလုံး ရှိနေမှာ ဖြစ်ပါတယ်။



Running Laravel Project

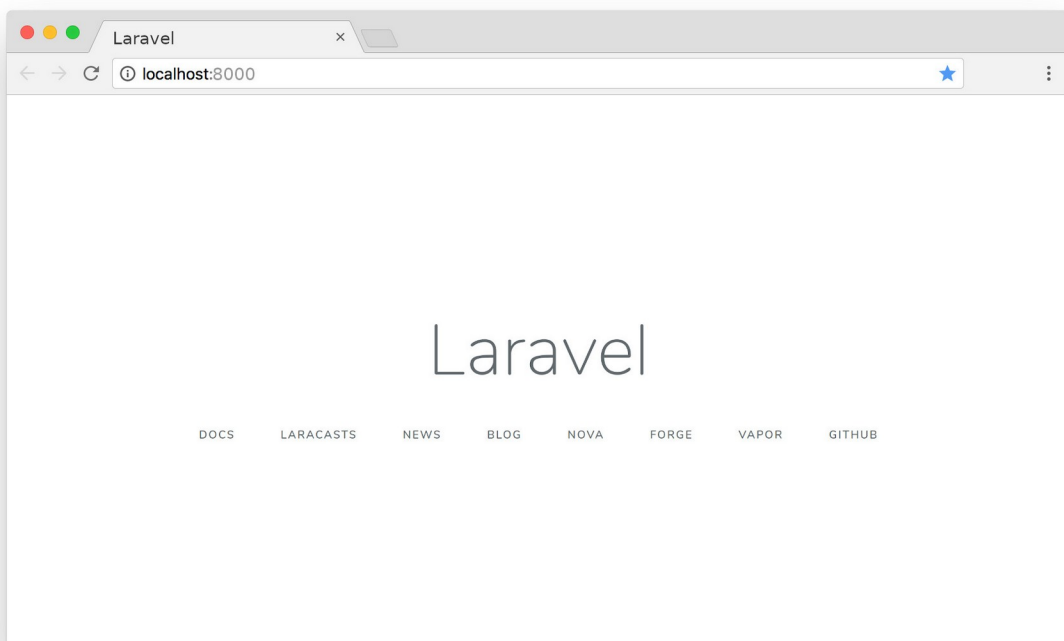
နောက်တစ်ဆင့်အနေနဲ့ တည်ဆောက်ထားတဲ့ blog ပရောဂျက်ဖိုဒါအတွင်းမှာ Terminal ကိုဖွင့်ပြီး ပရောဂျက်ကို အခုလို Run နိုင်ပါတယ်။

```
php artisan serve
```

ဒါဆိုရင် ပရောဂျက်ကို PHP Development Server ရဲ့အကူအညီနဲ့ Run သွားပြီဖြစ်လို့ Browser ကိုဖွင့်ပြီး အခုလို ရိုက်ထည့် စမ်းကြည့်လို့ရပါတယ်။

– <http://localhost:8000>

ရလဒ်ကတော့ အခုလိုဖြစ်မှာပါ။



ဒီအဆင့်ထိရသွားပြီဆိုရင် Laravel ကို အသုံးပြုပြီး ကုဒ်တွေရေးသားဖို့ အသင့်ဖြစ်သွားပါပြီ။

ဒီစာအုပ်ထဲမှာ နမူနာပရောဂျက်အနေနဲ့ Article တွေ ရေးတင်ပြီး Comment တွေ ပေးလို့ရတဲ့ Blog System လေးတစ်ခုကို တည်ဆောက်သွားကြမှာ ဖြစ်ပါတယ်။ လက်ရှိ ဒီစာရေးနေချိန် ထွက်ရှိထားတဲ့ Laravel နောက်ဆုံး Version က 7.9.2 ဖြစ်ပါတယ်။ ဒါကြောင့် နမူနာကုဒ်တွေဟာ Laravel 7.9 ကို အသုံးပြုဖော်ပြခြင်းဖြစ်ပြီး Version မတူရင် အနည်းငယ် ကွဲပြားမှု ရှိနိုင်တာကို သတိပြုပါ။

နောက်ထပ် ရှေ့ဆက်မသွားခင် ကြိုတင် အသိပေးချင်တာကတော့ ဒီစာအုပ်ဟာ လို - တို - ရှင်း စာအုပ်ဆို တဲ့အတိုင်း လိုရင်းကိုပဲ တိုက်ရိုက်ပြောသွားမှာ ဖြစ်ပါတယ်။ Laravel မှာ လိုချင်တဲ့ရလဒ်တစ်ခုရဖို့ ရေးသား နည်း သုံးလေးနည်း ရှိတတ်ပါတယ်။ အဲ့ဒါတွေအကုန် ထည့်ပြောနေမှာ မဟုတ်ဘဲ၊ သုံးသင့်တဲ့ တစ်နည်းထဲ ကိုပဲ ရွေးထုတ်ပြီး ပြောသွားမှာဖြစ်ပါတယ်။ အဲ့ဒီလို ပြောပြတဲ့ တစ်နည်းထဲနဲ့ပဲ အရင်ဆုံးပိုင်နိုင်အောင် လေ့လာလိုက်ပါ။ နောက်တော့မှ တစ်ခြားမူကွဲတွေ၊ တစ်ခြားနည်းလမ်းတွေကို လေ့လာပါ။ အစပိုင်းမှာ အဲ့ ဒီလို မူကွဲတွေကြောင့် သိပ်ပြီးတော့ ခေါင်းစားမခံပါနဲ့ဦးလို့ ပြောချင်ပါတယ်။

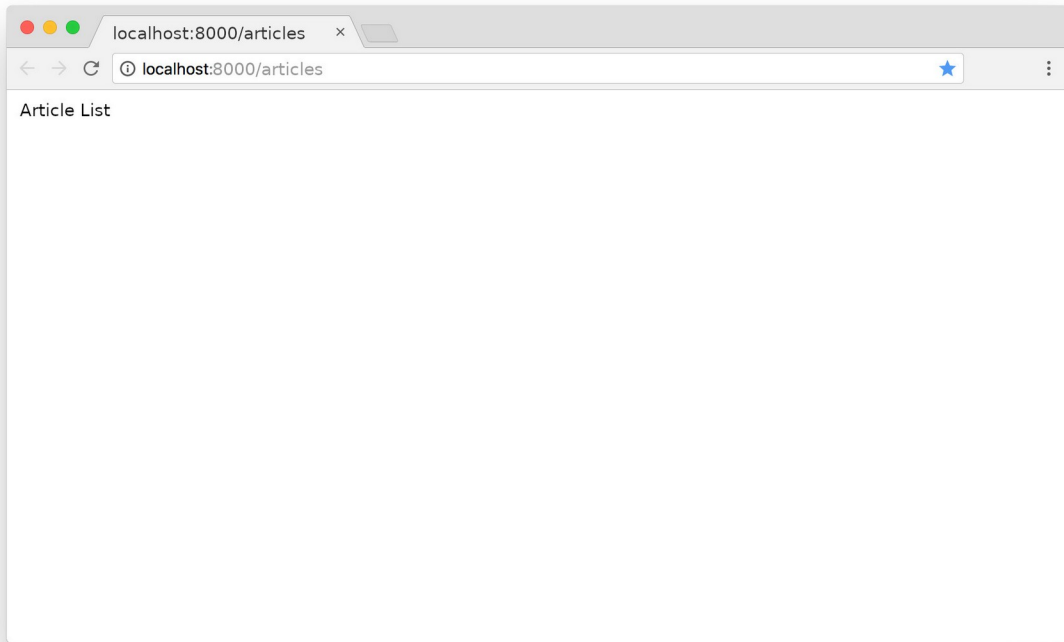
အခန်း (၄) - Routing

Laravel ပရောဂျက်နဲ့ပတ်သက်ရင် ပထမဆုံးအနေနဲ့ စတင်လေ့လာကြမှာကတော့ Routing ဖြစ်ပါတယ်။ Routing ကို မြန်မာလို လွယ်လွယ်ပြောရရင် လမ်းကြောင်းဆွဲခြင်း လို့ ပြောရမှာပါ။ ဘယ်လမ်းကိုသွားရင် ဘယ်ရောက်မလဲ သတ်မှတ်ပေးတဲ့သဘောပါ။ တစ်နည်းအားဖြင့် ဘယ် URL လိပ်စာကို သွားရင် ဘာ အလုပ်လုပ်ရမလဲ သတ်မှတ်ပေးခြင်း ဖြစ်ပါတယ်။ ပရောဂျက်ဖို့ဒါထဲက `/routes/web.php` ကို ဖွင့်ပြီး ဒီကုဒ်ကို ရေးဖြည့်ပြီး စမ်းကြည့်ပါ (သူ့နဂိုပါတဲ့ နမူနာကုဒ်ကို ဒီအတိုင်းထားပါ)။

```
Route::get('/articles', function () {  
    return 'Article List';  
});  
  
Route::get('/articles/detail', function () {  
    return 'Article Detail';  
});
```

Framework နဲ့အတူပါလာတဲ့ `get()` Route Method ကို အသုံးပြုထားခြင်း ဖြစ်ပါတယ်။ `get()` လို့ မျိုး တစ်ခြား Route Method တွေလည်းရှိပါသေးတယ်။ လောလောဆယ် `get()` နဲ့ `post()` နှစ်ခုမှတ်ထားရင် လုံလောက်ပါပြီ။ Basic API အခန်းရောက်တော့မှ တစ်ခြား Method တွေ ထပ်ကြည့်ပါဦးမယ်။ နမူနာမှာ `get()` Route Method အတွက် Parameter နှစ်ခုပေးထားပါတယ်။ ပထမတစ်ခုက URL လိပ်စာဖြစ်ပြီး ဒုတိယတစ်ခုက လုပ်ရမယ့်အလုပ် Function ဖြစ်ပါတယ်။ ဒီနမူနာအရ လိပ်စာက `/articles` ဆိုရင် သတ်မှတ်ထားတဲ့ Function အလုပ်လုပ်သွားတဲ့အတွက် Article List ဆိုတဲ့ စာတစ်ကြောင်းကို ပြန်ရမှာဖြစ်ပါတယ်။ အလားတူပဲ လိပ်စာက `/articles/detail` ဆိုရင်တော့ Article

Detail ဆိုတဲ့ စာတစ်ကြောင်းကို ပြန်ရမှာပါ။ ဒီလိုပါ -

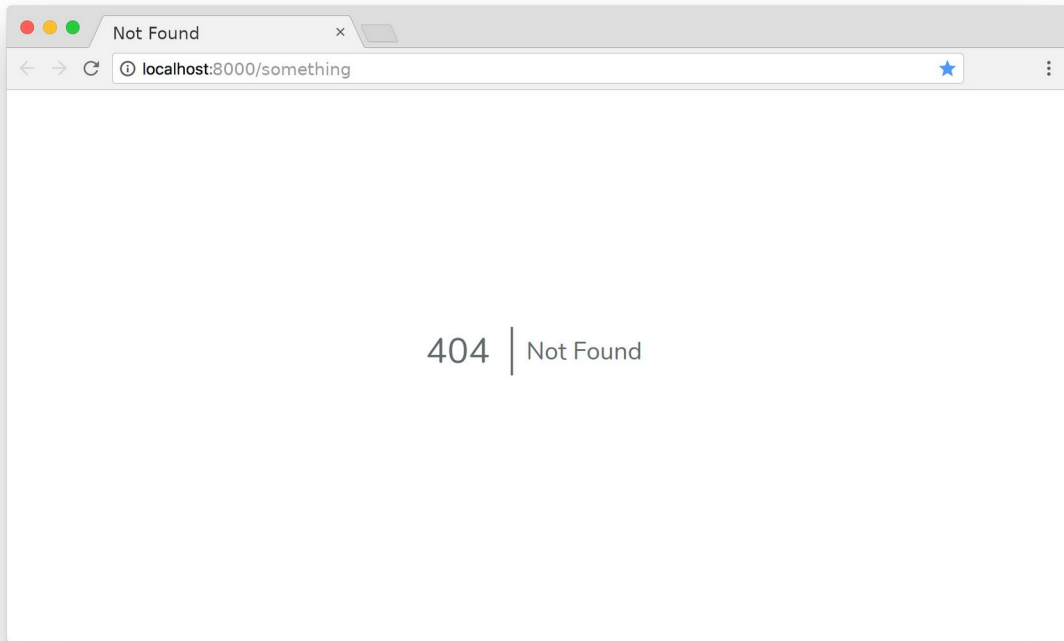


သတိပြုပါ - ဒီလို စမ်းကြည့်လို့ရဖို့အတွက် `php artisan serve` နဲ့ ပရောဂျက်ကို Run ထားဖို့ လိုအပ်ပါတယ်။ ပြီးတော့၊ နမူနာ Screenshot တွေပြတဲ့အခါ URL လိပ်စာနဲ့ ရလဒ်ကို တွဲတွဲပြီးတော့ ကြည့်ပေးပါ။

မရှိတဲ့လိပ်စာကို ရိုက်ထည့်ပြီး စမ်းကြည့်လို့လည်း ရပါတယ်။ ဥပမာ ဒီလိုရိုက်ထည့်ပြီး စမ်းကြည့်ပါ -

- <http://localhost:8000/something>

ရလဒ်ကအနေနဲ့ 404 Not Found ကို တွေ့မြင်ရမှာ ဖြစ်ပါတယ်။



Laravel မှာ Route လိုမျိုး လိုအပ်တဲ့အချိန်မှာ အလွယ်ယူသုံးလို့ရတဲ့ Class တွေ အများကြီး ရှိပါတယ်။ Facade Pattern လို့ခေါ်တဲ့ OOP Design Pattern တစ်ခုကို သုံးပြီးရေးထားတဲ့ Class တွေပါ။ သိပ်ခေါင်းစားမခံပါနဲ့။ အခေါ်အဝေါ်လောက်ပဲ မှတ်ထားပြီ နောက်မှ ဆက်လေ့လာပါ။ လောလောဆယ် အသင့်သုံးလို့ရတဲ့ Facade Class တွေ ရှိတယ်ဆိုတာလောက် မှတ်ထားရင် ရပါတယ်။ Source Code ထဲမှာ သွားကြည့်ချင်ရင် `/vendor/laravel/framework/src/illuminate/Support/Facade` ဆိုတဲ့ Path လမ်းကြောင်းနဲ့ ဖိုဒါတစ်ခု ရှိပါတယ်။ အဲ့ဒီထဲမှာ အသင့်သုံးလို့ရတဲ့ Facade Class တွေ ရှိနေပါတယ်။

ဒါပေမယ့် စောစောက နမူနာမှာ `Route::get()` လို့ရေးခဲ့ပေမယ့် `Route.php` ကုဒ်ဖိုင်ကို ဖွင့်ကြည့်ရင် `get()` Method ကို တွေ့ရမှာ မဟုတ်ပါဘူး။ `get()` Method တစ်ကယ်မရှိဘဲနဲ့ သုံးလို့ရအောင် Laravel က PHP ရဲ့ `__callStatic()` လို့ခေါ်တဲ့ Magic Method တစ်မျိုးကို သုံးထားပါတယ်။ အဲ့ဒါကြောင့် ခေါင်းစားစရာတွေလို့ ပြောတာပါ။ Laravel Framework ကို ဘယ်လိုတည်ဆောက်ထားတာလဲ သိချင်ရင် အဲ့ဒါမျိုးတွေကို လေ့လာရမှာပါ။ ဒါပေမယ့် ဒီ Framework ကို အသုံးပြုပြီး ပရောဂျက်တွေရေးချင်ယုံသက်သက်နဲ့တော့ ဒါတွေသိစရာမလိုသေးပါဘူး။ ဒါကြောင့် ဒီလိုအတွင်းပိုင်းကိစ္စတွေ နောက်ပိုင်းမှာ ထည့်မပြောတော့ပါဘူး။ ဒီစာအုပ်ရဲ့ ရည်ရွယ်ချက်က Laravel Framework ကို အသုံးပြုတတ်စေဖို့ ဖြစ်ပါ

တယ်။ ဒါကြောင့် သုံးနည်းပဲဆက်ပြောပါမယ်။ သုံးနည်းတတ်သွားပြီဆိုတော့မှ စိတ်ဝင်စားရင် ဘယ်လို တည်ဆောက်ထားတာလဲ ဆိုတာကို ဆက်လေ့လာရမှာပါ။

Dynamic Routes

Routing အကြောင်း ဆက်ပါမယ်။ Route လို့ပြောရင် Static Route နဲ့ Dynamic Route ဆိုပြီး နှစ်မျိုးရှိပါတယ်။ Static Route ဆိုတာကတော့ ပုံသေ သတ်မှတ်ထားတဲ့ လမ်းကြောင်းပါ။ Dynamic Route ကတော့ ပါလာတဲ့ Route Parameter ပေါ်မူတည်ပြီး ပြောင်းလဲနိုင်တဲ့ လမ်းကြောင်းပါ။ Dynamic Route တစ်ခုခုနမူနာကို စမ်းကြည့်နိုင်ဖို့ ဒီကုဒ်ကို `/routes/web.php` ထဲမှာ ထပ်ဖြည့်ရေးပါ။

```
Route::get('/articles/detail/{id}', function ( $id ) {
    return "Article Detail - $id";
});
```

သတိပြုစရာ (၃) ချက်ရှိပါတယ်။ ပထမတစ်ချက်က လိပ်စာမှာပါတဲ့ `{id}` ဖြစ်ပါတယ်။ Laravel မှာ Route Parameter ကို ဒီလိုပေးရပါတယ်။ အဓိပ္ပါယ်က `{id}` ဟာ အရှင်ဖြစ်ပြီး မိမိနှစ်သက်ရာတန်ဖိုးကို `{id}` နေရာမှာ ပေးလိုရတယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ ဥပမာ - `/articles/detail/123` ဆိုတဲ့ လိပ်စာ ဟာ ဒီသတ်မှတ်ချက်နဲ့ ကိုက်ညီသလို `/articles/detail/abc` ဆိုတဲ့လိပ်စာ ဟာလည်း ဒီ သတ်မှတ်ချက်နဲ့ ကိုက်ညီပါတယ်။

ဒုတိယတစ်ချက်ကတော့ Function Parameter ဖြစ်တဲ့ `$id` ပါ။ URL လိပ်စာရဲ့ `{id}` အဖြစ်ပေးလိုက်တဲ့ တန်ဖိုးကို Function ရဲ့ `$id` အဖြစ် Laravel က လက်ခံ အသုံးပြုပေးသွားမှာပါ။ တတိယအချက်ကတော့ PHP အခြေခံကို မေ့သွားမှာစိုးလို့သာ ထည့်ပြောတာပါ။ PHP မှာ String အတွင်း Variable တွေ တစ်ခါ တည်း ထည့်သွင်း အသုံးပြုလိုရင် Double Quote ကို အသုံးပြုရတယ် ဆိုတဲ့ အချက်ပါ။ ဒါကြောင့် `return` Statement အတွက် ပေးလိုက်တဲ့ String ကို Double Quote နဲ့ ရေးထားတာပါ။

ဒါကြောင့် လိပ်စာက `/articles/detail/123` ဆိုရင် ပြန်ရမှာက Article Detail - 123 ဖြစ်ပါတယ်။ `/articles/detail/abc` ဆိုရင်တော့ ပြန်ရမှာက Article Detail - abc ဖြစ်ပါတယ်။ စမ်းကြည့်နိုင်ပါ။ ဒီနည်းနဲ့ Laravel မှာ Dynamic Route တွေ သတ်မှတ် အသုံးပြုရပါတယ်။

Route Names

Routing နဲ့ပတ်သက်ပြီး နောက်ထပ်တစ်ခုအနေနဲ့ ထည့်သွင်းမှတ်သားသင့်တာကတော့ Route Name ဖြစ်ပါတယ်။ သတ်မှတ်ထားတဲ့ Route တွေကို အမည်ပေးထားခြင်းအားဖြင့် နောင်လိုအပ်တဲ့အခါ အလွယ်တစ်ကူ ပြန်လည်အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။ ဥပမာ -

```
Route::get('/articles/more', function() {
    return redirect('/articles/detail');
});
```

ဒီကုဒ်ရဲ့အဓိပ္ပါယ်က URL လိပ်စာ /articles/more ဖြစ်ခဲ့မယ်ဆိုရင် redirect() Helper Function ရဲ့ အကူအညီနဲ့ /articles/detail ကိုသွားခိုင်းလိုက်တာ ဖြစ်ပါတယ်။ /articles/detail ကို အခုလို အမည်ပေးထားနိုင်ပါတယ်။

```
Route::get('/articles/detail', function () {
    return 'Article Detail';
})->name('article.detail');
```

နောက်ဆုံးမှာ name() Method နဲ့ article.detail လို့ အမည်ပေးလိုက်တာပါ။ တစ်ကယ်တော့ အမည်က ကြိုက်သလို ပေးလို့ရပါတယ်။ တိုတိုတုတ်တုတ်ပဲ ပေးချင်လည်းရပါတယ်။ ဒါပေမယ့် Route Name တွေ များလာရင် တစ်ခုနဲ့တစ်ခုရောပြီး မှတ်ရခက်ကုန်မှာစိုးလို့ အလေ့အကျင့်ကောင်းတစ်ခု အနေနဲ့ အခုကတည်းက ပြည့်ပြည့်စုံစုံ ပေးထားတာပါ။ အခုလို အမည်ပေးထားပြီးပြီဖြစ်လို့ ဒီ Route ကိုလိုအပ်လို့ ပြန်သုံးချင်ရင် ပေးထားတဲ့ အမည်နဲ့ သုံးလို့ရပါပြီ။ ဥပမာ - စောစောက /articles/more Route လုပ်ဆောင် ချက်ကို အခုလို ပြင်လိုက်ပါမယ်။

```
Route::get('/articles/more', function() {
    return redirect()->route('article.detail');
});
```

အလုပ်လုပ်ပုံက အတူတူပါပဲ။ `/articles/more` ကိုသွားလိုက်ရင် `Redirect` လုပ်ထားတဲ့အတွက် `/articles/detail` ကိုပဲ ရောက်သွားမှာပါ။ ဒါပေမယ့် `URL` လိပ်စာကို မသုံးတော့ဘဲ ပေးထားတဲ့ `Route Name` ကို သုံးပြီး အလုပ်လုပ်လိုက်တာပါ။

ဆက်ပြောမယ်ဆိုရင် `Resource Route` တို့ `View Route` တို့ `Route Group` တို့ ကျန်ပါသေးတယ်။ `Resource Route` အကြောင်းကိုတော့ `Basic API` အခန်းကျမှ ဆက်ကြည့်ပါမယ်။

URL Pattern

`URL` လိပ်စာတွေ သတ်မှတ်တဲ့အခါ ကြိုက်တဲ့ပုံစံနဲ့ ကြိုက်သလိုပေးလို့ရပါတယ်။ ဒါပေမယ့် ပေးလို့ရတယ် ဆိုတိုင် စွတ်ပေးလို့ မဖြစ်ပါဘူး။ များလာတဲ့အခါ ကိုယ်ပေးထားတဲ့ `URL` ကို ကိုယ်မမှတ်မိတော့ဘဲ ရှုပ်ကုန် ပါလိမ့်မယ်။ `URL` လိပ်စာတွေပေးတဲ့အခါ လိုက်နာသင့်တဲ့ အချက်လေးတွေ ရှိပါတယ်။ လိုရင်းအနှစ်ချုပ် အနေနဲ့ ဒီ `Pattern` လေး (၂) ခုကို မှတ်ထားပေးပါ။

- `/resource/action/id`
- `/resource/action/id/sub-resource/sub-action`

စာလုံးအသေးတွေချည်းပဲ သုံးတယ်ဆိုတဲ့အချက်ကနေ စမှတ်ပါ။ လိုအပ်ရင် `Dash` ကို သုံးနိုင်ပါတယ်။ `Underscore` ကို မသုံးသင့်ပါဘူး။ `camelCase` တွေ မသုံးသင့်ပါဘူး။ ရှေ့ဆုံးက အချက်အလက် အမျိုးအစား (`Resource`) နဲ့ စသင့်ပါတယ်။ `Plural Case` (အများကိန်း) ဖြစ်သင့်ပါတယ်။ ဥပမာ - `articles, users, students, customers, products` စသဖြင့် လိုချင်တဲ့ အချက်အလက် အမျိုးအစား ဖြစ်ပါတယ် (`Noun` လို့လည်း ဆိုနိုင်ပါတယ်)။ သူ့နောက်က အလုပ်အမျိုးအစား (`Action`) လိုက်သင့်ပါတယ်။ ဥပမာ - `add, update, delete, view, detail` စသဖြင့် ဖြစ်ပါတယ် (`Verb` လို့လည်း ဆိုနိုင်ပါတယ်)။ နောက်ဆုံးကနေ `Unique Identifier (id)` လိုက်ရမှာပါ။ ဒီ `Pattern` အတိုင်း `URL` လိပ်စာ အချို့ကို နမူနာဖော်ပြလိုက်ပါတယ်။

- `/users`
- `/products/view/{id}`
- `/customers/update/{id}`
- `/students/add`

Sub Route ပါဝင်တဲ့ နမူနာတစ်ချို့ကိုလည်း ဖော်ပြပေးလိုက်ပါတယ်။

- /users/detail/{id}/photos
- /products/view/{id}/comments/add
- /students/show/{id}/marks

အတတ်နိုင်ဆုံး ဒီ Pattern ဘောင်ထဲမှာပဲ URL လိပ်စာတွေကို သတ်မှတ်ပေးပါ။ ဒီ Pattern ဘောင်ထဲမှာ ဝင်ဖို့ခက်တဲ့ ရှုပ်ထွေးတဲ့ လုပ်ဆောင်ချက်တွေ ရှိလာခဲ့ရင်တောင် ကြိုးစားပြီး ရအောင်သတ်မှတ်ပေးပါ။ အပိုထပ်ဆောင်းလိုအပ်တဲ့ အချက်အလက်တွေကို URL Query အနေနဲ့ ဖြည့်ပေးနိုင်ပါတယ်။ ဥပမာ - ရန်ကုန်တိုင်းမှာရှိတဲ့ ကျောင်းသူစာရင်းကို လိုချင်တယ်ဆိုကြပါစို့။ Resource (Noun) က students ပါ။ Action (Verb) က list (သို့) all ဖြစ်နိုင်ပါတယ်။ တစ်ကယ်တော့ list တို့ all တို့ကို Default လို့ သဘောထားပြီး မပေးတာ ပိုကောင်းပါတယ်။ ID ကတော့ ဒီနေရာမှာ မလိုအပ်ပါဘူး။ လိုချင်တဲ့ ရလဒ်မှာ ပါဝင်တဲ့ ရန်ကုန်တိုင်း တို့ ကျောင်းသူတို့က ဘာတွေလဲ။ Resource လား၊ Action လား၊ ID လား၊ Sub-Route လား။ တစ်ခုမှ မဟုတ်ပါဘူး။ ဒါကြောင့် ဒီလိုမျိုး မပေးသင့်ပါဘူး။

- /students/list/yangon/female

URL လေးက သုံးချင်စရာလေးပါ။ ဒါပေမယ့် Pattern ဘောင် မဝင်တော့ပါဘူး။ နောက်အလားတူ ကိစ္စမှာ ရှေ့နောက်မညီရင် ရှုပ်ကုန်ပါတော့မယ်။ ဥပမာ -

- /customers/all/male/yangon

ရှေ့နောက်အစီအစဉ်တွေ Consistence မဖြစ်တော့ပါဘူး။ အသုံးအနှုံးတွေ Consistence မဖြစ်တော့ပါဘူး။ Route တွေများလာတဲ့အခါ စီမံရက်ကုန်ပါလိမ့်မယ်။ ဒါကြောင့် /resource/action/id ဘောင်ထဲကနေသာ တူညီစွာပေးပါ။ လက်တွေ့လိုအပ်တဲ့အခါ URL Query တွေကို သုံးနိုင်ပါတယ်။ ဥပမာ

- /students?location=ygn&sex=female

ဒါဟာ မဖြစ်မနေလိုက်နာရမယ့် ပုံသေနည်းတော့ မဟုတ်ပါဘူး။ ဒါပေမယ့် လိုက်နာမယ်ဆိုရင် အကျိုးရှိတဲ့ URL Pattern Convention ဖြစ်ပါတယ်။

အခန်း (၅) - Model - View - Controller

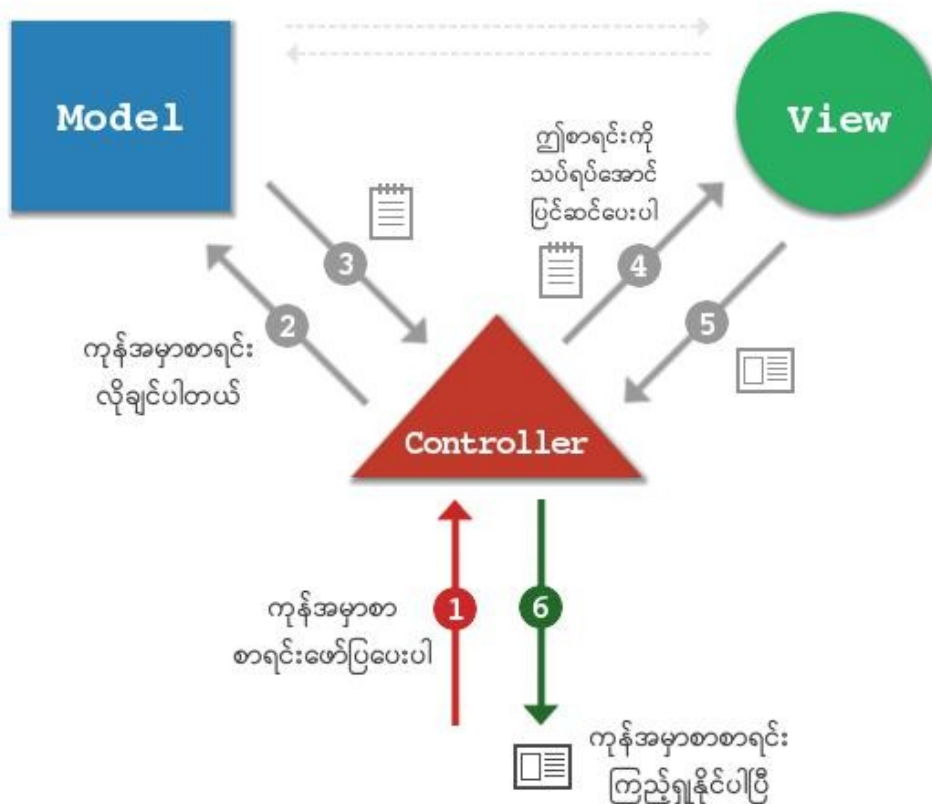
Laravel ဟာ MVC Framework တစ်ခုဖြစ်တဲ့အတွက် ရှေ့ဆက်မသွားခင် Model - View - Controller (MVC) အကြောင်းကို ကြားဖြတ်လေ့လာဖို့ လိုအပ်ပါတယ်။ MVC ဟာ ကနေ့ခေတ်မှာ Web Application Framework တိုင်းက စံထားပြီးသုံးနေကြတဲ့ Structure Pattern ဖြစ်ပါတယ်။ ဖတ်ရှုနားလည်ရလွယ်ကူပြီး ပြုပြင်ထိန်းသိမ်းရလည်း လွယ်ကူတဲ့ကုဒ်တွေ ရေးသားဖို့အတွက် အရေးပါတဲ့ နည်းစနစ်တစ်ခုပါ။ ဒါကြောင့် ဒီအကြောင်းက ချဲ့ရင်ချဲ့သလောက် ရပါတယ်။ ဒီစာအုပ်မှာတော့ ထုံးစံအတိုင်း ရှည်ရှည်ဝေးဝေးတွေကို ပြောမနေတော့ဘဲ လိုရင်းကိုပဲ ပြောမှာ ဖြစ်ပါတယ်။

MVC ရဲ့ M ဟာ Model ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ Model ဆိုတာ Data Model ကို ဆိုလိုပါတယ်။ ဒါကြောင့် Model ဆိုတာ Data တွေ စီမံခန့်ခွဲခြင်းလုပ်ငန်းလို့ ဆိုနိုင်ပါတယ်။ MVC ရဲ့ V ကတော့ View ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ User Interface လို့လည်း ဆိုနိုင်ပါတယ်။ MVC ရဲ့ အဓိကလိုရင်းကတော့ ကုဒ်တွေရေးတဲ့ အခါမှာ Data စီမံတဲ့ကုဒ် (Model) နဲ့ UI စီမံတဲ့ကုဒ် (View) ကို သီးခြားစီ ခွဲခြားပြီးရေးသားရခြင်း ပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် MVC နဲ့ ပတ်သက်ရင် ဒီသတ်မှတ်ချက်က ကြပ်ကြပ်မတ်မတ်လိုက်နာပေးရမယ့် သတ်မှတ်ချက် ဖြစ်ပါတယ်။ Model ကုဒ်နဲ့ View ကုဒ်ကို ဘယ်တော့မှ ရောမရေးပါနဲ့။

MVC ရဲ့ C ကတော့ Controller ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ သူ့ရဲ့လုပ်ငန်းကတော့ User Input နဲ့ Output တွေကို စီမံတဲ့လုပ်ငန်းဖြစ်ပါတယ်။ Web Application တစ်ခုမှာ User Input တွေဟာ Request အနေနဲ့လာပြီး Output တွေကို Response အနေနဲ့ ပြန်ထုတ်ပေးရပါတယ်။ ဒါကြောင့် တစ်နည်းအားဖြင့် Controller ဆိုတာ Request နဲ့ Response တွေကို စီမံတဲ့လုပ်ငန်း လို့ ဆိုနိုင်ပါတယ်။

ဒါဟာ MVC အနှစ်ချုပ်ပါပဲ။ ကုဒ်တွေရေးတဲ့အခါ ဒီအပိုင်း (၃) ပိုင်းကို သီးခြားစီခွဲခြားပြီး ရေးသားရမှာ ဖြစ်ပါတယ်။ ဘယ်လိုခွဲရမှာလည်းဆိုတာကို ခေါင်းစားစရာမလိုပါဘူး။ သဘောသဘာဝ သိထားရင် ရပါပြီ။ ကိုယ်တိုင်အသေးစိတ် လိုက်စီမံစရာ မလိုပါဘူး။ Laravel က ကုဒ်တွေရေးတဲ့အခါ MVC Pattern နဲ့ကိုက်ညီအောင် ရေးသားနိုင်ဖို့အတွက် ထည့်သွင်း စီစဉ်ပေးထားပြီးဖြစ်ပါတယ်။ ကိုယ့်ဘက်က သဘောသဘာဝကို သိထားပြီးပြီဆိုရင် ကုဒ်ရေးတာကတော့ Framework က စီစဉ်ပေးထားတဲ့အတိုင်း ရေးယုံပါပဲ။

MVC Pattern ကို အသုံးပြုရေးသားထားတဲ့ ကုဒ်ရဲ့ User Input (Request) ကို စတင်လက်ခံတဲ့ အဆင့်ကနေ နောက်ဆုံး Output (Response) ကို User ထံပြန်ပေးတဲ့အဆင့်ထိ သွားလေ့ရှိတဲ့ Workflow ကို ပုံလေးနဲ့လည်း ဖော်ပြပေးလိုက်ပါတယ်။



ဒီပုံဟာ Professional Web Developer လို့ခေါ်တဲ့ ကျွန်တော်ရေးသားထားတဲ့ စာအုပ်မှာ အသုံးပြုထားတဲ့ ပုံဖြစ်ပါတယ်။ များလေးတွေမှာ နံပါတ်စဉ်တပ်ပေးထားလို့ အစီအစဉ်အတိုင်း ကြည့်သွားပါ။

MVC Pattern နဲ့ ရေးထားတဲ့ ကုဒ်ရဲ့ အလုပ်လုပ်ပုံအဆင့်ဆင့်ကို ပုံမှာပြထားတဲ့အတိုင်း သေချာနားလည်အောင် ကြည့်ပြီး ခေါင်းထဲမှာ စွဲနေအောင် မှတ်ထားဖို့ လိုပါတယ်။ ဒီတော့မှ ကိုယ်ရေးတဲ့ ကုဒ်ရဲ့ အလုပ်လုပ်ပုံအဆင့်ဆင့်ကို ဆက်စပ်ပြီး မြင်နိုင်စွမ်း ရှိမှာဖြစ်ပါတယ်။

Professional Web Developer စာအုပ်ကို အောက်ပါလိပ်စာမှာ Download ရယူနိုင်ပါတယ်။ အဲ့ဒီစာအုပ်မှာ ပါတဲ့ MVC သင်ခန်းစာနဲ့လည်း ယှဉ်တွဲပြီး ကြည့်သင့်ပါတယ်။

- <https://eimaung.com/professional-web-developer/>

အခန်း (၆) - Controller

ပြီးခဲ့အခန်းမှာ Controller ရဲ့အလုပ်က Request, Response တွေစီမံဖို့လို့ ပြောခဲ့ပါတယ်။ Request တွေ စီမံတယ်ဆိုတာ -

1. User ပေးပို့တဲ့ Request Data တွေကို လက်ခံမယ်၊ စီစစ်မယ်၊
2. ပြီးရင် Model တို့ View တို့နဲ့ ဆက်သွယ်ပြီး လုပ်စရာရှိတာလုပ်မယ်၊
3. ပြီးတဲ့အခါ နောက်ဆုံးရလဒ်ကို User ကို Response အနေနဲ့ ပြန်ပေးမယ်၊

- ဆိုတဲ့ အလုပ်ပါ။ ဒီနေရာမှာ လုပ်စရာရှိတာကို Model တို့ View တို့နဲ့ ဆက်သွယ်ပြီး လုပ်တယ်ဆိုတာကို သတိပြုပါ။ သူ့ကိုယ်တိုင် မလုပ်ပါဘူး။ သူ့အလုပ်၊ သူ့တာဝန်က Request ကိုလက်ခံပြီး Response ကို ပြန် ပေးဖို့သာ ဖြစ်ပါတယ်။

Controller ကုန်တွေ စတင်ရေးသားစမ်းသပ်နိုင်ဖို့အတွက် Controller ဖိုင် တည်ဆောက်ပေးရပါမယ်။ Laravel မှာ ဒီလို လိုအပ်တဲ့ ကုန်ဖိုင်တွေကို တည်ဆောက်ပေးနိုင်တဲ့ Code Generator အသင့် ပါဝင်ပါတယ်။ ဒါကြောင့် Controller ဖိုင်တည်ဆောက်တဲ့အလုပ်ကို ကိုယ့်ဘာသာ လုပ်စရာ မလိုပါဘူး။ ပရောဂျက် ဖိုဒါထဲမှာ ဒီ Command ကို Run ပေးပြီး Controller ဖိုင်တစ်ခု တည်ဆောက်နိုင်ပါတယ်။

```
php artisan make:controller ArticleController
```

artisan ဆိုတာ Laravel Framework နဲ့အတူပါဝင်လာတဲ့ ပရောဂျက်ကို စီမံနိုင်တဲ့ နည်းပညာပါ။ အဲ့ဒီ

artisan ရဲ့အကူအညီနဲ့ `make:controller`, `make:model` စသဖြင့် လိုအပ်တဲ့ ကုဒ်ဖိုင်တွေ တည်ဆောက်နိုင်ပါတယ်။ ပေးထားတဲ့နမူနာမှာ `make:controller` ကိုသုံးပြီး Controller ဖိုင် တည်ဆောက်ထားပါတယ်။ နောက်ဆုံးက `ArticleController` ဆိုတာကတော့ Controller Class ရဲ့အမည်ပါ။ Class Name ဖြစ်တဲ့အတွက် CapitalCase နဲ့ပေးရပြီး နမူနာမှာပေးထားသလို `__Controller` ဆိုတဲ့ Suffix နဲ့ အဆုံးသတ်ပေးသင့်ပါတယ်။ ဥပမာ - `ArticleController`, `CommentController`, `CategoryController` စသဖြင့်ပါ။

ဒီ Command ကို Run လိုက်တာနဲ့ `ArticleController.php` ဆိုတဲ့အမည်နဲ့ ဖိုင်တစ်ခု `/app/Http/Controllers` ဖိုဒါထဲမှာ တည်ဆောက်ပေးသွားပါလိမ့်မယ်။ အထဲမှာလည်း အခုလို ကုဒ်တွေ တစ်ခါတည်း ပါဝင်လာပါလိမ့်မယ်။

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ArticleController extends Controller
{
    //
}
```

ထိပ်ဆုံးမှာ `namespace` ကြေညာချက်ကိုသတိထားကြည့်ရင် အခန်း (၂) မှာ ပြောခဲ့သလို ဖိုဒါ Path လမ်းကြောင်းအတိုင်း Namespace ကို ပေးထားတာကို တွေ့ရနိုင်ပါတယ်။

Controller ရဲ့တာဝန်ဟာ Request/Response တွေတာဝန်ယူဖို့လို့ ခဏခဏ ပြောဖြစ်ပါတယ်။ User Request ဆိုတာ URL ကနေတစ်ဆင့်လာမှာ ဖြစ်လို့ Controller ဟာ Route နဲ့ တွဲပြီး အလုပ်လုပ်ဖို့လိုပါတယ်။ URL လမ်းကြောင်းက Route မှာ သတ်မှတ်ထားတာမို့လို့ပါ။ ဒီသဘောသဘာဝ ကို စမ်းသပ်နိုင်ဖို့ အတွက် `ArticleController.php` ထဲက ကုဒ်ကို အခုလို ဖြည့်စွက်လိုက်ပါ။

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ArticleController extends Controller
{
    public function index()
    {
        return "Controller - Article List";
    }

    public function detail($id)
    {
        return "Controller - Article Detail - $id";
    }
}
```

index() နဲ့ detail() ဆိုပြီး Method နှစ်ခု ရေးပေးလိုက်တာပါ။ ပြီးတဲ့အခါ routes/web.php မှာ Route တွေကို အခုလို ပြင်ပေးရပါမယ်။

```
Route::get('/articles', 'ArticleController@index');
Route::get('/articles/detail/{id}', 'ArticleController@detail');
```

ပြီးခဲ့တဲ့အခန်းမှာ Route တွေသတ်မှတ်တဲ့အခါ ရှေ့က URL လိပ်စာပေးပြီး နောက်က အလုပ်လုပ်ရမယ့် Function ကိုရေးပေးခဲ့တာပါ။ အခုတော့ ပြောင်းသွားပါပြီ။ ရှေ့က URL လိပ်စာဖြစ်ပြီး နောက်က အလုပ်လုပ်ရမယ့် Controller Method ဖြစ်သွားပါပြီ။ ဒီနည်းနဲ့ Route နဲ့ Controller ကို ချိတ်ဆက်ပေးနိုင်ပါတယ်။ လက်ရှိ routes/web.php ရဲ့ ကုဒ်အပြည့်အစုံက ဒီလိုဖြစ်သင့်ပါတယ်။

```
<?php

use Illuminate\Support\Facades\Route;

Route::get('/', 'ArticleController@index');
Route::get('/articles', 'ArticleController@index');
Route::get('/articles/detail/{id}', 'ArticleController@detail');
```

စမ်းထားတဲ့ ကုဒ်တွေနဲ့ Comment တွေကို ဖယ်ထုတ်လိုက်တာပါ။ စုစုပေါင်း Route (၃) ခုပဲ ရှိပါတယ်။ ဘာမှမပါတဲ့ URL အလွတ်ဆိုရင်လည်း ArticleController ရဲ့ index() Method ကိုပဲ အလုပ်လုပ်ဖို့ ညွှန်းပေးထားတာကို သတိပြုပါ။ နဂိုပါလာတဲ့ကုဒ်က Welcome View ကိုပြတဲ့ကုဒ်ဖြစ်ပြီး အခုတော့ အဲ့ဒီ Welcome View ကို မလိုချင်လို့ ဖယ်လိုက်တာပါ။ လက်တွေ့စမ်းကြည့်ပါ။ URL လိပ်စာတစ်ခု ထည့်သွင်းပေးလိုက်ရင် ညွှန်းဆိုထားတဲ့ Controller Method အလုပ်လုပ်သွားတယ်ဆိုတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

Controller မှာလည်း Single Action Controller တို့ Resource Controller တို့လို လုပ်ဆောင်ချက်တွေ ကျန်ပါသေးတယ်။ အခုထည့်ကြည့်စရာ မလိုသေးပါဘူး။ ထည့်ကြည့်သင့်တဲ့ တစ်ချက်ကတော့ Controller တွေများလာရင် /app/Http/Controllers ထဲမှာ ဖိုင်တွေ ပွထမနေအောင် ဖိဒါလေးတွေခွဲပြီး ထားနည်းဖြစ်ပါတယ်။ ဥပမာ - Article ဖိဒါအောက်မှာ ArticleController, Comment Controller, CategoryController တို့ကို ထားပြီး Product ဖိဒါအောက်မှာ Product Controller, ReviewController, CategoryController စသဖြင့် သက်ဆိုင်ရာ Controller တွေကို အုပ်စုခွဲပြီး ဖိဒါလေးတွေ ခွဲထားတဲ့ သဘောမျိုးပါ။ ဒီအတွက်လည်း လွယ်ပါတယ်။ Controller ဖိုင်တည်ဆောက်စဉ်မှာ အခုလို ဆောက်ပေးလိုက်ယုံပါပဲ။

```
php artisan make:controller Product/ProductController
```

ဒီလိုဆိုရင် Laravel က ProductController.php ဖိုင်ကို /app/Http/Controllers/Product အောက်မှာ တည်ဆောက်ပေးသွားမှာ ဖြစ်ပါတယ်။ ဒီ Controller ထဲက Method တွေကို Route နဲ့ချိတ်လိုရင်တော့ အခုလို ချိတ်ပေးရမှာပါ။

```
Route::get('/products', 'Product/ProductController@index');
```

အလုပ်လုပ်ရမယ့် Controller Method နေရာမှာ Namespace Path ကို မှန်အောင်ထည့်ပေးရခြင်းပဲ ဖြစ်ပါတယ်။ ဒီနည်းနဲ့ Controller တွေကို သူဖို့ဒါနဲ့သူ Organize လုပ်ထားလို့ ရပါတယ်။

အခန်း (၇) - View

လက်ရှိ Route နဲ့ Controller အတွက် ရေးစမ်းတဲ့ကုဒ်တွေမှာ URL လိပ်စာတစ်ခုကို လက်ခံရရှိရင် စာကြောင်း တစ်ကြောင်းကို Response အနေနဲ့ ပြန်ပေးဖို့ ရေးခဲ့တာပါ။ ဒီအခန်းမှာတော့ Response ကို အဲ့ဒီလို စာတစ်ကြောင်းအနေနဲ့ မဟုတ်တော့ဘဲ HTML Template အနေနဲ့ ပြန်ပေးပုံ ပေးနည်းကို လေ့လာ ကြမှာဖြစ်ပါတယ်။ ဒီအတွက် View ကို အသုံးပြုနိုင်ပါတယ်။ Laravel မှာ View Template တွေကို `/resources/views` ဖိုဒါထဲမှာ ရေးပေးရပါတယ်။ View Template ကုဒ်ဖိုင်တွေ တည်ဆောက်ပေးတဲ့ Code Generator တော့ Laravel မှာ တစ်ခါတည်း မပါပါဘူး။ Third-party Package အနေနဲ့ ထပ်ထည့် လို့ရပေမယ့် မထည့်တော့ပါဘူး။ ကိုယ်ဘာသာပဲ ဖိုင်ကိုတည်ဆောက်ပါမယ်။ ဒါကြောင့် `/resources/views` ဖိုဒါအောက်မှာ `/articles` ဆိုတဲ့ အမည်နဲ့ ဖိုဒါတစ်ခု ထပ်ဆောက်ပါ။ ကိုယ်တိုင် ရေးသားမယ့် View Template တွေကို အဲ့ဒီ `/articles` ဖိုဒါထဲမှာသိမ်းသွားမှာပါ။

မှတ်ချက် - ဖိုင်တွေ ဖိုဒါတွေကို `s`, `es` နဲ့ အဆုံးသတ်ပြီး Plural Case နဲ့ အမည်ပေးတဲ့ ဖိုင်တွေ ဖိုဒါတွေ ရှိပါတယ်။ သတိထားပါ။ အကြောင်းမဲ့ မဟုတ်ပါဘူး။ ပေးသင့်လို့ ပေးခဲ့တာပါ။ Convention Over Configuration ကြောင့် ပေးတာရှိသလို Coding Standard ခေါ် အများစံထားပြီး ပေးလေ့ရှိတဲ့ နည်းမို့လို့ ပေးတာတွေလည်း ရှိပါတယ်။ တစ်ခုမကျန် လိုက်ရှင်းပြနေရင် ပိုရှုပ်ပြီး ပေရှည်နေမှာစိုးလို့ အကုန်မရှင်း တော့တာပါ။ ကျေးဇူးပြုပြီး အဲ့ဒီ `s`, `es` လေးတွေကို သတိထားပြီး နမူနာမှာ ပေးတဲ့အတိုင်းပဲ ကြိုးစား ပြီး လိုက်ပေးဖို့ သတိထားပေးပါ။ Laravel လေ့လာစလူတွေ ကုဒ် Error ဖြစ်နေရင် အများအားဖြင့် အဲ့ဒီ နာမည်လေးတွေ လွဲနေလို့ တက်ကြတာ တော်တော်များပါတယ်။ Error တက်နေရင် အဲ့ဒီ နာမည်တွေကို အရင် ပြန်စစ်ကြည့်ပါ။

/resources/views/articles ဖိုဒါအောက်မှာ index.blade.php အမည်နဲ့ ဖိုင်တစ်ခု တည်ဆောက်ပေးပါ။ ဖိုင် Extension က blade.php ဖြစ်ရပါမယ်။ Laravel က Blade လို့ခေါ်တဲ့ Template နည်းပညာတစ်ခုကို သုံးထားတဲ့အတွက်ကြောင့်ပါ။ ဒီနည်းပညာအကြောင်းကို ကြားဖြတ် လေ့လာစရာ မလိုသေးပါဘူး။ လိုအပ်လာတော့မှ ထူးခြားချက်လေးတွေ ရွေးမှတ်လိုက်ရင် ရပါပြီ။ Symfony Framework ကသုံးတဲ့ Twig Template နည်းပညာတို့ Smarty Template နည်းပညာတို့ ဆိုရင် မရပါဘူး။ သီးခြား Language လို ဖြစ်နေလို့ အချိန်ပေးပြီးလေ့လာဖို့ လိုနိုင်ပါတယ်။ Blade ကတော့ သပ်သပ် အချိန်ပေးလေ့လာနေစရာမလိုပါဘူး။ အများအားဖြင့် PHP ရေးထုံးအတိုင်းပဲ ရေးရလို့ လွယ်ကူပါ တယ်။ တည်ဆောက်ထားတဲ့ index.blade.php ထဲမှာ ဒီ HTML ကုဒ်ကို ရေးပေးပါ။

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
  <meta charset="utf-8">
  <title>Article List</title>
</head>
<body>
  <h1>Article List</h1>
  <ul>
    <li>Article One</li>
    <li>Article Two</li>
  </ul>
</body>
</html>
```

ဘာမှဆန်းပြားတဲ့ကုဒ်တွေ မပါဝင်တဲ့ ရိုးရိုး HTML ကုဒ်ဖြစ်ပါတယ်။ ပြီးတဲ့အခါ Article Controller ရဲ့ index() Method ကို အခုလိုပြင်ပေးပါ (Controller ဖိုင်တည်နေရာတော့ ထပ်မ ပြောတော့ပါဘူး၊ မှတ်မိဦးမယ် ထင်ပါတယ်)။

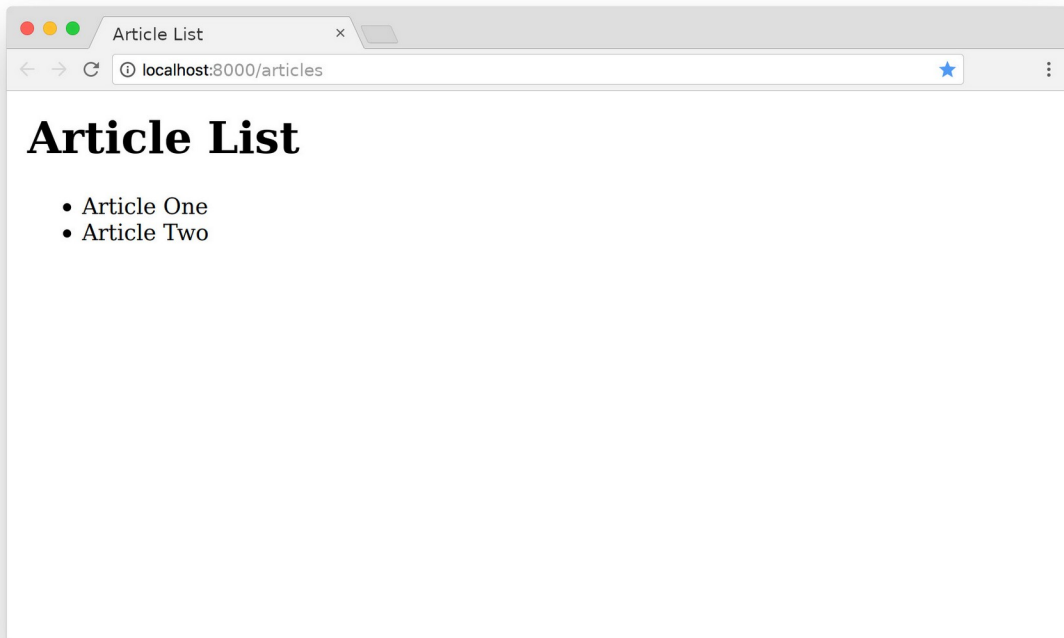
```
public function index()
{
    return view('articles/index');
}
```

ဒီတစ်ခါ index() Method က စာတစ်ကြောင်းကို ပြန်မပေးတော့ဘဲ view() Function ရဲ့ အကူအညီ နဲ့ articles ဖိုဒါထဲက index Template ကို Response အနေနဲ့ ပြန်ပေးလိုက်ခြင်း ဖြစ်ပါတယ်။

နောက်တစ်နည်းအနေနဲ့ အခုလို ရေးနိုင်ပါတယ်။

```
public function index()  
{  
    return view('articles.index');  
}
```

အတူတူပါပဲ။ Template Path ပေးတဲ့နေရာမှာ Slash အစား Dot ကို ပြောင်းသုံးလိုက်တာပါ။ Laravel က ဒီလိုရေးတာကို လက်ခံပါတယ်။ ဖတ်ရတာ မျက်စိထဲမှာ ပိုရှင်းတဲ့အတွက် ဒီလို Dot နဲ့ရေးတာကို ပိုပြီး တော့ လူကြိုက်များပါတယ်။ ခုနေ စမ်းကြည့်ရင် ရလဒ်က အခုလို ဖြစ်မှာပါ။



နမူနာပုံကိုလေ့လာကြည့်လိုက်ရင် URL လိပ်စာက /articles ဖြစ်နေတဲ့အခါ ကျွန်တော်တို့ရေး ပေးလိုက်တဲ့ index Template ကို တွေ့မြင်ရမှာပါ။ URL /articles ဆိုရင် Article Controller@index Method အလုပ်လုပ်သွားပါတယ်။ Route မှာ သတ်မှတ်ထားလို့ပါ။ ArticleController ရဲ့ index() Method က view() Function ကိုသုံးပြီး index Template ကို ပြန်ပေးလို့ အခုလိုရလဒ်ကို ရရှိခြင်းပဲ ဖြစ်ပါတယ်။

ဆက်လက်ပြီး Controller ကနေ View ကို Data ပေးပို့ပေးနည်း လေ့လာကြပါမယ်။ Article Controller ရဲ့ `index()` Method ကို အခုလိုပြင်ပေးပါ။

```
public function index()
{
    $data = [
        [ "id" => 1, "title" => "First Article" ],
        [ "id" => 2, "title" => "Second Article" ],
    ];

    return view('articles.index', [
        'articles' => $data
    ]);
}
```

အတွေ့အကြုံအရ လေ့လာစလူတွေ ဒီနားလေးမှာ အရမ်းမျက်စိလည်ကြလို့ သတိထားကြည့်ပေးပါ။ `$data` Variable ဟာ နမူနာ Array တစ်ခုဖြစ်ပါတယ်။ `view()` Function မှာ Parameter နှစ်ခုဖြစ်သွားပါပြီ။ ပထမတစ်ခုက Template ဖြစ်ပြီး ဒုတိယတစ်ခုကတော့ Data ဖြစ်ပါတယ်။ Array Format နဲ့ ပေးရပါတယ်။ `'articles' => $data` လို့ ရေးထားတဲ့အတွက် `$data` ကို `articles` အနေနဲ့ ပေးလိုက်တာပါ။ ဒီလိုပေးလိုက်တဲ့အတွက် Template မှာ `$articles` Variable ကို သုံးလို့ရသွားပါလိမ့်မယ်။

နောက်တစ်ခေါက် ပြန်ပြောပါဦးမယ်။ ပေးလိုက်တာက `$data` ကို ပေးလိုက်တာပါ။ အသုံးပြုတဲ့အခါ `$articles` လို့ အသုံးပြုပေးရမှာ ဖြစ်ပါတယ်။ `'articles' => $data` လို့ ပြောထားတဲ့ အတွက် ကြောင့်ပါ။

ဆက်လက်ပြီး `/articles/index.blade.php` ဖိုင်ထဲကတုဒ်ကို အခုလိုပြင်ပေးပါ။

```

<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
    <meta charset="utf-8">
    <title>Article List</title>
</head>
<body>
    <h1>Article List</h1>
    <ul>
        <?php foreach($articles as $article): ?>
            <li><?php echo $article['title'] ?></li>
        <?php endforeach ?>
    </ul>
</body>
</html>

```

\$articles ကို PHP foreach() နဲ့ Loop ပါတ်ပြီး title တွေကို ရိုက်ထုတ်ဖော်ပြထားတာပါ။ အဓိပ္ပါယ်က Controller ကပေးလိုက်တဲ့ Data ကို View က အသုံးပြုနေတဲ့ သဘောပဲ ဖြစ်ပါတယ်။ ဒါလေးကို ကောင်းကောင်း သဘောပေါက်ဖို့ အရေးကြီးပါတယ်။

နမူနာမှာရေးတဲ့အခါ ရိုးရိုး PHP ရေးထုံးကိုသာ သုံးပါတယ်။ Blade Template နည်းပညာကို အသုံးပြုပြီး ပြင်ရေးကြည့်ပါမယ်။ ဒီလိုရေးရမှာပါ -

```

<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
    <meta charset="utf-8">
    <title>Article List</title>
</head>
<body>
    <h1>Article List</h1>
    <ul>
        @foreach($articles as $article)
            <li>{{ $article['title'] }}</li>
        @endforeach
    </ul>
</body>
</html>

```

Operator နှစ်ခုပဲ မှတ်ရမှာပါ။ @ နဲ့ {{ }} ဖြစ်ပါတယ်။ @ သင်္ကေတကို ရိုးရိုး PHP Statement တွေ

အတွက် သုံးရပြီး `{{ }}` အဖွင့်အပိတ်သင်္ကေတကို `echo` နဲ့ `Output` ရိုက်ထုတ်တဲ့ `Statement` တွေ အတွက် သုံးရတယ်လို့ အလွယ်မှတ်နိုင်ပါတယ်။ ဒါကြောင့် `<?php ?>` အဖွင့်အပိတ်တွေ `echo` တွေ မလိုအပ်တော့ပါဘူး။ ရေးရတာရှင်းသွားသလို ဖတ်လို့လည်း ပိုကောင်းသွားပါတယ်။ မှတ်ရတာလည်း မများသလို `XSS Escape` လို့ လုံခြုံရေးအတွက် အရေးပါတဲ့ လုပ်ဆောင်ချက်တွေကိုလည်း ကိုယ်သိလိုက်စရာမလိုဘဲ တစ်ခါတည်း ထည့်လုပ်ပေးသွားလို့ ဒီရေးနည်းကိုသာ ဆက်လက် အသုံးပြုသွားသင့်ပါတယ်။

ဆက်ပြောမယ်ဆိုရင် `Master Template` တို့ `View Composer` တို့အကြောင်း ပြောရမှာပါ။ မပြောသေးပါဘူး။ ဒီအခန်းမှာ ဒီလောက်ပဲ မှတ်ထားပါ။ နောက်ပိုင်းမှာ `Master Template` အပါအဝင် `UI` နဲ့ `Template` ပိုင်း လုပ်စရာတွေ အများကြီး ကျန်ပါသေးတယ်။

အခန်း (၈) - Migration and Model

အစီအစဉ်အရ ဆက်ကြည့်ရမှာက Model အကြောင်းဖြစ်ပါတယ်။ ဒါပေမယ့် Model အကြောင်းမသွားခင် Database နဲ့ပတ်သက်တဲ့ အကြောင်းကို အရင်သွားရပါမယ်။ ပထမဆုံး MySQL Database တစ်ခုတည်ဆောက်လိုက်ပါ။ Database အမည်ကို ကြိုက်သလို ပေးလို့ရပါတယ်။ ပရောဂျက်အမည်နဲ့ ကိုက်သွားအောင် `laravel_blog` လို့ ပေးလိုက်ပါမယ်။ phpMyAdmin အသုံးပြုတတ်သူက phpMyAdmin ကို အသုံးပြုနိုင်ပါတယ်။ ဒီ Command ကို အသုံးပြုပြီး တော့လည်း တည်ဆောက်နိုင်ပါတယ်။

```
mysql -u root
```

MySQL Shell ကို Username `root` နဲ့ ဝင်ရောက်ခြင်းဖြစ်ပါတယ်။ ဒီလို ဝင်ရောက်နိုင်ဖို့ MySQL Database Server ကို Run ထားပြီး ဖြစ်ဖို့တော့လိုပါတယ်။ အကယ်၍ ကိုယ့်စက်ထဲက MySQL Database မှာ Password ရှိရင်တော့ ဒီလို ဝင်ရမှာပါ။

```
mysql -u root -p
```

Password လာတောင်းတဲ့အခါ ပေးလိုက်ရင် ရပါပြီ။ MySQL Shell ထဲကို ရောက်ပြီဆိုရင် ဒီ Query ကို Run ပြီး Database တည်ဆောက်နိုင်ပါတယ်။

```
CREATE DATABASE laravel_blog
```

ပြီးရင် `exit` နဲ့ ပြန်ထွက်လိုက်လို့ ရပါပြီ။ Database တည်ဆောက်ဖို့ပဲလိုပါတယ်။ Table တည်ဆောက်တဲ့ ကိစ္စတွေ၊ Data ထည့်သွင်းတဲ့ ကိစ္စတွေကို Laravel ကုဒ် ဘက်ကနေ ဆက်လုပ်သွားမှာပါ။

Database Setting

ပြီးတဲ့အခါ Database Name, Username, Password စတဲ့အချက် တွေကို Laravel က သိသွားအောင် သတ်မှတ်ပေးရပါဦးမယ်။ ပရောဂျက် ဖိုဒါထဲမှာ `.env` အမည်နဲ့ ဖိုင်တစ်ခုပါဝင်ပါတယ်။ Database Setting တွေ အပါအဝင် Environment Setting တွေကို အဲ့ဒီဖိုင်ထဲမှာ စုစည်းပြီး ရေးသားထားပါတယ်။ Database နဲ့ ပက်သက်တာက ဒီအပိုင်းပါ -

```
...
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
...
```

အခုလို ပြင်ပေးရမှာ ဖြစ်ပါတယ် -

```
...
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_blog
DB_USERNAME=root
DB_PASSWORD=
...
```

`DB_DATABASE` တစ်ခုပဲ ပြင်ပေးလိုက်တာပါ။ ကျန်တာတွေက မလိုအပ်လို့ မပြင်ထားပါဘူး။ အကယ်၍ စာဖတ်သူရဲ့ စက်မှ Database Password တွေဘာတွေ သတ်မှတ်ထားရင်တော့ ဒီနေရာမှာ တစ်ခါတည်း မှန်အောင်ပြင်ပေးရမှာ ဖြစ်ပါတယ်။

Migration

Laravel မှာ Database Table တွေကို စီမံဖို့အတွက် Migration လို့ခေါ်တဲ့ နည်းပညာပါဝင်ပါတယ်။ Migration ကို Table Structure စီမံတဲ့ နည်းပညာလို့ အလွယ်မှတ်နိုင်ပါတယ်။ Migration နဲ့ Model ကုန် ဖိုင်တွေကို အခုလို တည်ဆောက်လို့ရပါတယ်။

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

ဒီ Command ရဲ့ အဓိပ္ပါယ်က Article အမည်နဲ့ Model ဖိုင်တစ်ခု တည်ဆောက်ခြင်း ဖြစ်ပါတယ်။ ထုံးစံအတိုင်း Class Name ကို CapitalCase နဲ့ပေးပါတယ်။ ရှေ့နောက်မှ ဘာမှမတွဲပါဘူး။ ဥပမာ - ArticleModel, CommentModel စသဖြင့် ပေးစရာမလိုပါဘူး။

အရေးကြီးတာက ဟိုးနောက်ဆုံးက -m လေးဖြစ်ပါတယ်။ အဲ့ဒါ Migration ဖိုင်တစ်ခါတည်း တည်ဆောက် ပေးစေဖို့အတွက် ထည့်ပေးလိုက်တာပါ။ တစ်ကယ်တန်းက make:migration နဲ့ make:model ဆို ပြီးနှစ်ခါ Run ပေးရမှာပါ။ မ Run ချင်လို့ တစ်ကြောင်းတည်းနဲ့ အခုလို Model ရော Migration ကိုပါ တွဲ Run လိုက်တဲ့ သဘောပါ။

ဒီ Command ကို Run လိုက်တဲ့အတွက် /app/Article.php ဆိုတဲ့ Model ဖိုင်နဲ့ /database/migrations/xxx_create_articles_table.php ဆိုတဲ့ဖိုင်နှစ်ခု တည်ဆောက်ပေးသွားမှာ ဖြစ်ပါတယ်။ xxx နေရာမှာ Run တဲ့အချိန်နဲ့ ရက်စွဲပေါ်မူတည်ပြီး တစ်ယောက်နဲ့တစ်ယောက် မတူဘဲ ကွဲပြားသွားမှာပါ။

ကြားဖြတ်ပြီး တစ်ခုပြောချင်ပါတယ်။ Model နဲ့ Migration အမည်ဆင်တူလို့ အမှတ်မမှားပါနဲ့။ Migration ဆိုတာ Table ကို စီမံဖို့ဖြစ်ပြီး Model ဆိုတာ (Table ထဲက) Data ကို စီမံဖို့ ဖြစ်တယ် လို့ အလွယ်မှတ်နိုင် ပါတယ်။ Data တွေ စီမံဖို့အတွက် ပုံမှန်အားဖြင့် SQL Query တွေကို အသုံးပြုရပါမယ်။ Laravel မှာတော့ Eloquent လို့ခေါ်တဲ့ ORM နည်းပညာ တစ်ခုပါဝင်ပြီး ဒီနည်းပညာကို ရှေ့ဆက်အသုံးပြုသွားမှာပါ။ ORM ဆိုတာ Object Relational Mapping ရဲ့ အတိုကောက်ဖြစ်ပြီး လိုရင်းအနှစ်ချုပ်ကတော့ Database Data တွေကို SQL Query တွေမသုံးဘဲ OOP ကုန်နဲ့ စီမံလို့ရအောင် ကြားခံဆောင်ရွက်ပေးတဲ့ နည်းပညာ လို့ မှတ်နိုင်ပါတယ်။ PHP မှာ Entity, Doctrine, Eloquent စသဖြင့် ORM နည်းပညာ အမျိုးမျိုးရှိပါတယ်။

ဆက်လက်ပြီး Table တည်ဆောက်ဖို့အတွက် တည်ဆောက်လိုတဲ့ Table ရဲ့ ဖွဲ့စည်းပုံကို သတ်မှတ်ပေးပါမယ်။ /database/migrations/xxx_create_articles_table.php ဖိုင်ကိုဖွင့်လိုက်ပါ။ အထဲမှာ အခုလိုကုန်တွေ ပါဝင်ပါလိမ့်မယ်။

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
}
```

up() Method အတွင်းမှာ Table တည်ဆောက်တဲ့ကုန်ကို တစ်ခါတည်း ရေးထားပေးပါတယ်။ Table ရဲ့ အမည်ကို articles လို့ ပေးထားပါတယ်။ လောလောဆယ်မှာ id နဲ့ timestamps ဆိုတဲ့ သတ်မှတ်ချက်နှစ်ခု ပါဝင်ပါတယ်။ ဒါကြောင့်ခုနစ် ဒီကုန်ကို Run ရင် id, created_at, updated_at ဆိုတဲ့ Column (၃) ခု ပါဝင်တဲ့ articles Table ကိုရရှိမှာဖြစ်ပါတယ်။ Laravel မှာ timestamps အတွက် created_at နဲ့ updated_at ကို သုံးလို့ပါ။ မ Run ခင် ပါဝင်စေလိုတဲ့ Column တွေ အရင်

ထည့်ပေးလိုက်ပါမယ်။ `up()` Method ကို အခုလို ပြင်ပေးလိုက်ပါ။

```
public function up()
{
    Schema::create('articles', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('body');
        $table->integer('category_id');
        $table->timestamps();
    });
}
```

`title` အမည်နဲ့ `String Column` တစ်ခု၊ `body` အမည်နဲ့ `Text Column` တစ်ခု၊ `category_id` အမည်နဲ့ `Integer Column` တစ်ခု၊ စုစုပေါင်း `Column (၃)` ခုထပ်တိုးပေးလိုက်တာပါ။ ပြီးရင် ဒီကုဒ်ကို `Run` လို့ရပါပြီ။ ဒီလိုပါ -

```
php artisan migrate
```

ဒါဆိုရင် `id`, `title`, `body`, `category_id`, `created_at`, `updated_at` ဆိုတဲ့ `Column (၅)` ခုပါဝင်တဲ့ `articles Table` ကို တည်ဆောက်သွားမှာ ဖြစ်ပါတယ်။ သတိပြုရမှာကတော့ `Framework` နဲ့အတူ `Migration` ဖိုင် (၂) ခု တစ်ခါတည်းပါလာပါသေးတယ်။ `/database/migrations` ဖိုဒါထဲမှာ ကြည့်လို့ရပါတယ်။ ဒါကြောင့် ကိုယ်တိုင် `Migration` ဖိုင်တစ်ခု ထည့်လိုက်ပေမယ့် တစ်ကယ်တမ်း `Run` သွားတာ `Migration` ဖိုင် (၃) ခုဖြစ်တယ်ဆိုတဲ့ အချက်ကို သတိပြုပါ။

Database Seeding

`Table` တည်ဆောက်ပြီးတဲ့အခါ နမူနာ `Data` တစ်ချို့ထည့်ပေးဖို့ လိုအပ်တတ်ပါတယ်။ `Laravel` မှာ ဒီလို နမူနာ `Data` ထည့်ပေးနိုင်တဲ့ နည်းပညာလည်း တစ်ခါတည်း ပါဝင်ပါတယ်။ `Database Seeding` လို့ခေါ်ပါတယ်။ နမူနာ `Data` တွေထည့်သွင်းဖို့အတွက် `Model Factory` လို့ခေါ်တဲ့ နမူနာ `Data (Model Object)` တွေ ထုတ်ပေးနိုင်တဲ့ကုဒ်ကို အရင်ရေးသင့်ပါတယ်။ ဒီလိုပါ -

```
php artisan make:factory ArticleFactory
```

ဒီ Command ကို Run လိုက်ရင် ArticleFactory.php အမည်နဲ့ ဖိုင်တစ်ခု /databases/factories ထဲမှာ တည်ဆောက်ပေးသွားပါလိမ့်မယ်။ အထဲမှာ အခုလို ကုဒ်တွေ ပါဝင်မှာပါ။

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\Model;
use Faker\Generator as Faker;

$factory->define(Model::class, function (Faker $faker) {
    return [
        //
    ];
});
```

Faker လို့ခေါ်တဲ့ Random Sample Data ထုတ်ပေးနိုင်တဲ့ နည်းပညာတစ်ခုကို Import လုပ်ထားတာကို သတိပြုပါ။ ပြီးရင် ဒီ Model Factory က ပြန်ပေးတဲ့ Model မှာ ပါဝင်ရမယ့် Property တွေကို သတ်မှတ်ပါမယ်။ ဒါကြောင့် ကုဒ်ကို အခုလိုပြင်ပေးဖို့ လိုအပ်ပါတယ်။

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\Article;
use Faker\Generator as Faker;

$factory->define(Article::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
        'category_id' => rand(1, 5),
    ];
});
```

ပထမဆုံး Import လုပ်ထားတဲ့ App\Model အစား App\Article ဖြစ်သွားတာကို သတိပြုပါ။ ပြီး

တော့ `$factory->define()` အတွက်ပေးထားတာလည်း `Article::class` ဖြစ်သွားပါပြီ။

`Article::class` ဆိုတာ `'App\Article'` ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ PHP Namespace Resolution ခေါ် Namespace Path အပြည့်အစုံကို လိုချင်ရင် `::class` ကို သုံးရတာပါ။

ဒီ Factory က ပြန်ပေးမှာကတော့ စောစောက Migration နဲ့ Table တည်ဆောက်စဉ်က ပေးလိုက်တဲ့ `title`, `body`, `category_id` ဆိုတဲ့ Column သုံးခုအတွက် တန်ဖိုးတွေကို ပြန်ပေးမှာပါ။ `title` အတွက် Sample စာတစ်ကြောင်း၊ `body` အတွက် Sample စာတစ်ပိုဒ်နဲ့ `category_id` အတွက် 1, 5 ကြား Random တန်ဖိုး တစ်ခုတို့ကို ပေးလိုက်တာ ဖြစ်ပါတယ်။ Faker က `name`, `phoneNo`, `email`, `address` စသဖြင့် အသုံးဝင်တဲ့ Random Sample တွေ ထုတ်ပေးနိုင်ပါတယ်။ အခုတော့ `sentence` နဲ့ `paragraph` ကို အသုံးပြုထားပါတယ်။ Faker မှာပါတဲ့ လုပ်ဆောင်ချက် အပြည့်အစုံကို သိချင်ရင် အောက်ကလိပ်စာမှာ လေ့လာနိုင်ပါတယ်။

- <https://github.com/fzaninotto/Faker>

ပြီးတဲ့အခါ `/database/seeds` မှာရှိတဲ့ `DatabaseSeeder.php` ဆိုတဲ့ဖိုင်ကို ဖွင့်ကြည့်ပါ။ အထဲမှာ ဒီလိုကုန်တွေ ရှိနေတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

```
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UserSeeder::class);
    }
}
```

`run()` Method ထဲမှာ Factory ရဲ့ အကူအညီနဲ့ နမူနာ Data တွေ Table ထဲကို ထည့်ပေးတဲ့ကုဒ်ကို အခု လို ရေးပေးရမှာ ဖြစ်ပါတယ်။

```
public function run()
{
    factory(App\Article::class, 20)->create();
}
```

ဒါဟာ စောစောကတည်ဆောက်လိုက်တဲ့ `ArticleFactory` က ပြန်ပေးတဲ့ အချက်အလက်တွေကိုသုံးပြီး Model အခု (၂၀) တည်ဆောက်လိုက်တဲ့သဘောပါ။ တစ်နည်းအားဖြင့် Record အကြောင်း (၂၀) ထည့်သွင်းလိုက်ခြင်းပဲ ဖြစ်ပါတယ်။ ဒီကုဒ်ကို Run ပေးဖို့တော့လိုပါသေးတယ်။ Run ပေးလိုက်မှာ Record တွေက ဝင်သွားမှာပါ။ ဒီလို Run ရပါတယ် -

```
php artisan db:seed
```

Database Table နဲ့ Sample Data တွေကို Manual တစ်ခုချင်းလုပ်စရာမလိုဘဲ အခုလို ကုဒ်လေးရေးပြီး Run ယုံနဲ့ စီမံလို့ရတယ်ဆိုတာ လက်တွေ့မှာ တော်တော်အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တွေပါ။ အထူးသဖြင့် အများနဲ့ပူးပေါင်းအလုပ်လုပ်တဲ့အခါ တစ်ယောက်က Table Structure ပြင်လိုက်လို့ ကိုယ့်ဆီမှာ ကုဒ်ကအလုပ်မလုပ်တော့ဘူး ဆိုတာမျိုး ဖြစ်စရာမလိုတော့ပါဘူး။ သူထည့်ပေးလိုက်တဲ့ Migration ကုဒ်ကို Run လိုက်ယုံနဲ့ Updated Structure ကို ကိုယ့်ဆီမှာလည်း ရသွားမှာ ဖြစ်ပါတယ်။

Model

အခုဆိုရင် Table တည်ဆောက်ခြင်း၊ Sample Data ထည့်သွင်းခြင်းတွေ ပြီးသွားပြီမို့လို့ ဒီ Data ကို အသုံးပြုတော့မယ်။ `ArticleController.php` မှိုက်ကိုဖွင့်ပြီး အခုလိုပြင်ပေးပါ။

```
<?php

namespace App\Http\Controllers;

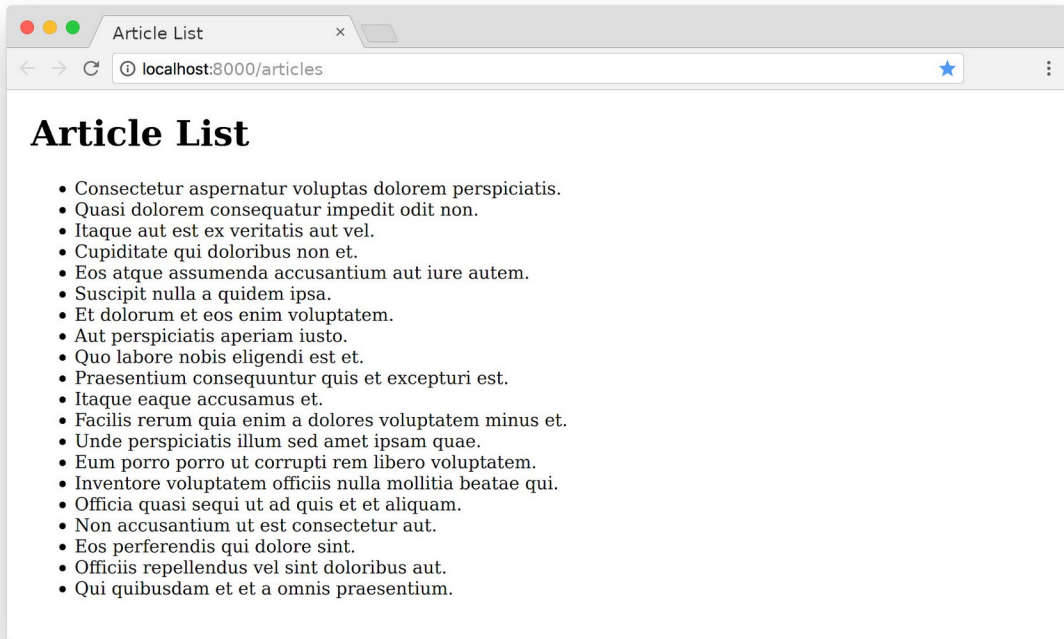
use App\Article;
use Illuminate\Http\Request;

class ArticleController extends Controller
{
    public function index()
    {
        $data = Article::all();

        return view('articles.index', [
            'articles' => $data
        ]);
    }

    public function detail($id)
    {
        return "Controller - Article Detail - $id";
    }
}
```

ထိပ်နားမှာ ကျွန်တော်တို့တည်ဆောက်ထားတဲ့ Model ဖြစ်တဲ့ `App\Article` ကို Import လုပ်ထားတာ သတိပြုပါ။ ပြီးတဲ့အခါ `Article::all()` Method ကိုသုံးပြီး `articles` Table ထဲကရှိ Record အားလုံးကို ထုတ်ယူထားပါတယ်။ အရမ်းလွယ်ပါတယ်။ `SELECT * FROM articles` စသဖြင့် SQL Query တွေ မလိုအပ်တော့ပဲ သတ်မှတ်ထားတဲ့ Method တွေ Property တွေကနေတစ်ဆင့် Data ကို အခုလို စီမံနိုင်ခြင်း ဖြစ်ပါတယ်။ စမ်းကြည့်လိုက်ရင် ရလဒ်က ဒီလိုဖြစ်မှာပါ။



ဒါဟာ `articles` Table ထဲက Data တွေကို တွေ့မြင်နေရခြင်းပဲ ဖြစ်ပါတယ်။

ဒီအခန်းမှာတော့ ဒီလောက်နဲ့ ခဏနားကြပါဦးစို့။ နောက်အခန်းတွေမှာ Model နဲ့ ပတ်သက်တဲ့ ကိစ္စတွေ ဆက်လေ့လာစရာ ကျန်ရှိပါသေးတယ်။

အခန်း (၉) - Authentication

Model နဲ့ပက်သက်ပြီး ကျန်နေတာတွေ ဆက်မကြည့်ခင် UI ပိုင်း သပ်သပ်ရပ်ရပ်ဖြစ်စေဖို့ကိုလည်း တစ်ခါတည်း တွဲပြီး လုပ်သွားပါဦးမယ်။ ဒီအတွက် Laravel မှာ Bootstrap CSS Framework ကို အသုံးပြု ဖန်တီးထားတဲ့ UI တစ်ခါတည်း ပါပါတယ်။ ပြီးတော့ User Login, Register, Logout စတဲ့ Authentication (Auth) လုပ်ဆောင်ချက်တွေလည်း ကြိုတင်ရေးပေးထားပြီးသာ ပါပါတယ်။

အရင် Laravel Version အဟောင်းတွေမှာဆိုရင် UI နဲ့ Auth က အတွဲလိုက်ပါ။ `make:auth` ကို အသုံးပြုပြီး Authentication ကုဒ်မိုင်တွေ တည်ဆောက်လိုက်တာနဲ့ UI ပါ တစ်ခါတည်း ပါသွားတာပါ။ အခုနောက်ပိုင်း Version တွေမှာတော့ UI ကသပ်သပ် Auth က သပ်သပ်ဖြစ်သွားပါပြီ။ သပ်သပ်စီ သုံးလို့ ရသလို တွဲသုံးလို့လည်း ရပါတယ်။ ဒါပေမယ့် အသုံးပြုလိုရင် `laravel/ui` Package ကို Install လုပ် ပေးဖို့ လိုသွားပါတယ်။ တစ်ခါတည်း တွဲထည့်မပေးတော့ပါဘူး။ ဒါကြောင့် `laravel/ui` ကို အခုလို Install လုပ်ပေးပါ။

```
composer require laravel/ui
```

လက်ရှိ Laravel ပရောဂျက်ဖိုဒါယ်မှာ Run ရမှာပါ။ ဒီနည်းနဲ့ ထပ်မံထည့်သွင်း အသုံးပြုလိုတဲ့ တစ်ခြား PHP Package တွေကိုလည်း ထပ်ထည့်လို့ ရပါတယ်။ `laravel/ui` ကို Install လုပ်ပြီးပြီဆိုရင် Auth နဲ့ UI အတွက် ကုဒ်တွေကို အခုလို အတွဲလိုက် ဖန်တီးယူလို့ရပါပြီ။

```
php artisan ui bootstrap --auth
```

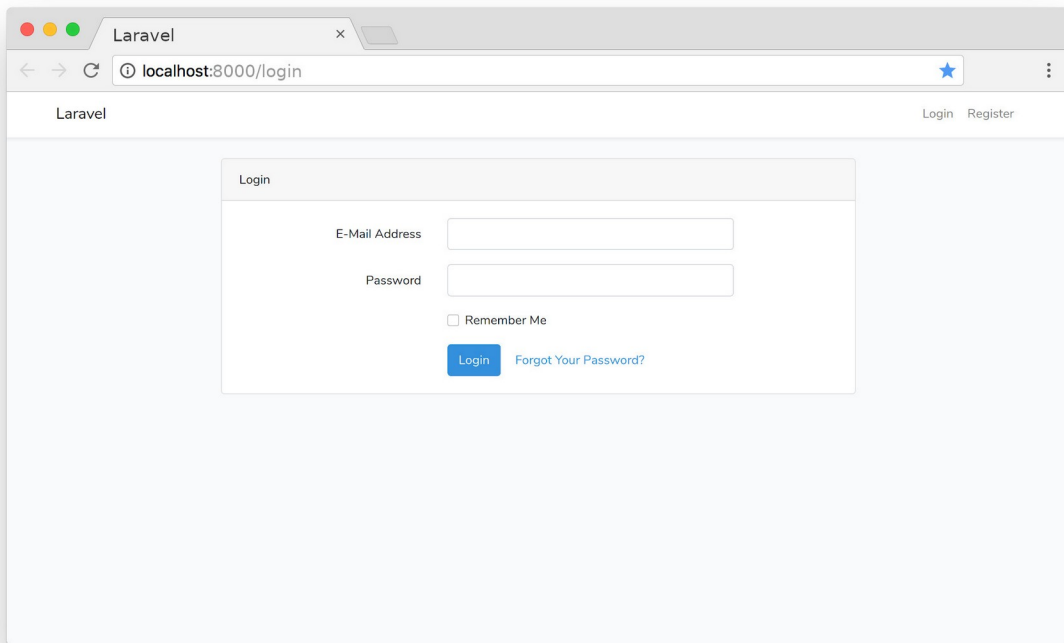
UI မှာ (၃) မျိုးရှိပါတယ်။ `vue`, `react` နဲ့ `bootstrap` ပါ။ ဒီစာအုပ်မှာတော့ `Bootstrap` ကိုပဲ သုံးသွားမှာပါ။ အပေါ်က `Command` က လိုအပ်တဲ့ဖိုင်တွေ ဆောက်ပေးသွားတယ်။ ပြီးတော့ `Bootstrap` ကို သုံးမယ်လို့ သတ်မှတ်လိုက်ပါတယ်။ တစ်ကယ့် `Bootstrap CSS Framework` ဖိုင်တွေကိုတော့ `Download` မလုပ်ရသေးပါဘူး။ ဒါကြောင့် ဒီ `Command` တွေ ဆက် `Run` ပေးဖို့ လိုပါသေးတယ်။

```
npm install
npm run dev
```

ဒီ `Command` တွေက `Composer` မဟုတ်တော့ပါဘူး။ `NPM` ဖြစ်သွားပါပြီ။ `Frontend Package` တွေနဲ့ `JavaScript Package` တွေအတွက် `NPM` ကိုပဲ သုံးကြလေ့ရှိတာပါ။ ဒါကြောင့် ကိုယ့်စက်ထဲမှာ `NPM` ကို `Install` လုပ်ထားပြီး ဖြစ်ဖို့တော့ လိုပါသေးတယ်။ မရှိသေးရင် nodejs.org ကနေ `Node` ကို `Download` လုပ်ပြီး `Install` လုပ်လိုက်ရင် `NPM` လည်း တစ်ခါတည်း ပါဝင်သွားပါလိမ့်မယ်။

ပထမ `Command` က `Bootstrap` အပါအဝင် လိုအပ်တဲ့ `Frontend Library` တွေကို `Download` လုပ်ပေးသွားမှာဖြစ်ပြီး၊ ဒုတိယ `Command` ကတော့ `CSS` ကုဒ်တွေ `JavaScript` ကုဒ်တွေကို အသင့်အသုံးပြုနိုင်အောင် `Compile` လုပ်ပေးမှာ ဖြစ်ပါတယ်။

ဒီ `Command` တွေအားလုံး `Run` ပြီးဖို့ လိုအပ်တဲ့ဖိုင်တွေ စုံပြီဆိုရင်တော့ `/login`, `/register` စတဲ့ `URL` တွေနဲ့ စမ်းသပ်အသုံးပြုလို့ရသွားပါပြီ။



ပြီးခဲ့တဲ့ အခန်းမှာ Migration ကုဒ်တွေကို Run တဲ့အခါ User Data တွေသိမ်းဖို့အတွက် လိုအပ်တဲ့ Table လည်း တစ်ခါတည်း ပါဝင်သွားပြီး ဖြစ်ပါတယ်။ ဒါကြောင့် User Account တွေဆောက်၊ Login တွေဝင်ပြီး တော့ စမ်းမယ်ဆိုရင်လည်း စမ်းကြည့်လို့ရတာကို တွေ့ရမှာဖြစ်ပါတယ်။

အခန်း (၁၀) – Master Template

အခုဆိုရင် UI Package ကိုလည်း ထည့်သွင်းပြီးဖြစ်လို့ ကျွန်တော်တို့ ရေးသားလက်စကုဒ်ကို သပ်သပ်ရပ်ရပ်ဖြစ်အောင် ပြင်ကြပါမယ်။ View Template တွေ ရေးတဲ့အခါ တူညီတဲ့ကုဒ်ကို ထပ်ခါထပ်ခါ ရေးစရာ မလိုဘဲ တစ်ကြိမ်ရေးထားပြီး လိုအပ်တဲ့နေရာကနေ ပြန်ယူသုံးလို့ရတဲ့နည်းတွေ ရှိပါတယ်။ အဲ့ဒီထဲက အရေးအကြီးဆုံးနည်း တစ်ခုကတော့ Master Template ဖြစ်ပါတယ်။

Master Template တွေကို ကိုယ်တိုင်ရေးရင်လည်း ရနိုင်ပေမယ့် UI Package ကို ထည့်သွင်းလိုက်တဲ့အတွက် Master Template တစ်ခု အသင့်ပါဝင်သွားလို့ အခုတော့ အဲ့ဒီ Template ကို ဆက်လက်အသုံးပြုပြီး ရေးသားသွားမှာပါ။ တည်နေရာကတော့ `/resources/views/layouts/app.blade.php` ဖြစ်ပါတယ်။ ကျွန်တော်တို့ရေးသားလက်စဖြစ်တဲ့ `/resources/views/articles` ဖိုဒါထဲက `index.blade.php` ကို အခုလို ပြင်ရပါမယ်။

```

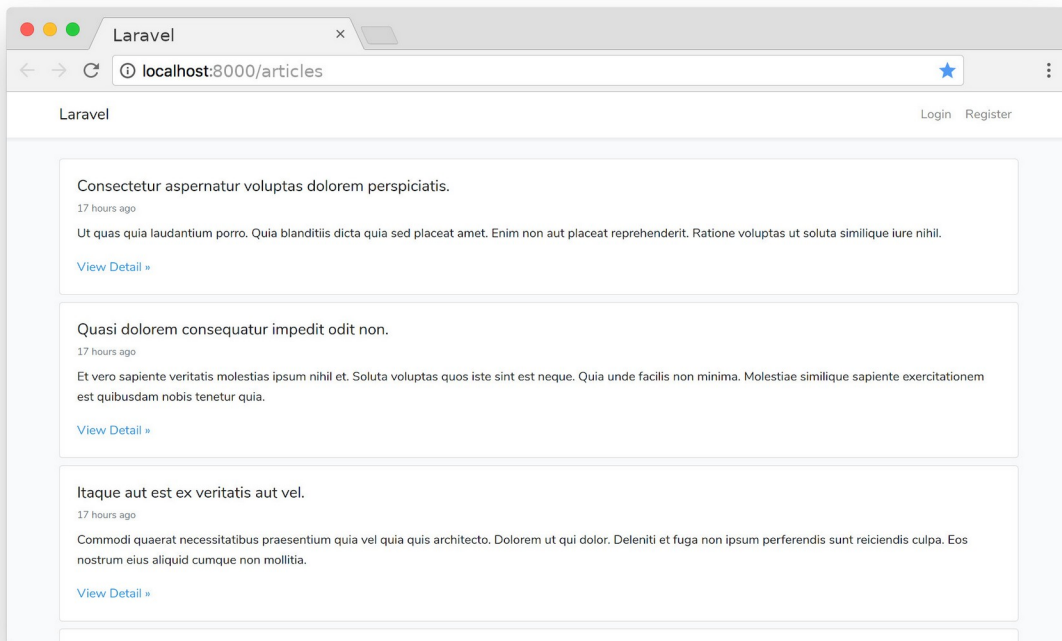
@extends("layouts.app")

@section("content")
    <div class="container">
        @foreach($articles as $article)
            <div class="card mb-2">
                <div class="card-body">
                    <h5 class="card-title">{{ $article->title }}</h5>
                    <div class="card-subtitle mb-2 text-muted small">
                        {{ $article->created_at->diffForHumans() }}
                    </div>
                    <p class="card-text">{{ $article->body }}</p>
                    <a class="card-link"
                        href="{{ url("/articles/detail/$article->id") }}">
                        View Detail &raquo;
                    </a>
                </div>
            </div>
        @endforeach
    </div>
@endsection

```

ပထမဆုံးအနေနဲ့ Blade ရဲ့ @extends() ကိုသုံးပြီး layouts/app ကို လှမ်းယူလိုက်ပါတယ်။ ပြီးတော့မှ အဲ့ဒီ Layout ထဲမှာ Content အနေနဲ့ ဖော်ပြရမယ့် Template ကုဒ်ကို @section("content") ကိုသုံးပြီး ရေးပေးလိုက်တာပါ။ ဒါကြောင့် layouts/app Template ထဲမှာ ပေးလိုက်တဲ့ Content နဲ့ ပြပေးတဲ့ ရလဒ်ကို ရမှာ ဖြစ်ပါတယ်။

Content အနေနဲ့ Bootstrap CSS Framework ရဲ့ လုပ်ဆောင်ချက်တွေကို အသုံးပြုပြီး title, created_at နဲ့ body တို့ကို ဖော်ပြထားပါတယ်။ ထူးခြားချက်အနေနဲ့ created_at ပေါ်မှာ diffForHumans() Method ကို သုံးထားလို့ ရက်စွဲအချိန်ကို 3 seconds ago, 1 day ago စသဖြင့် ဖတ်ရှုနားလည်ရ လွယ်ကူတဲ့ပုံစံနဲ့ ဖော်ပြပေးမှာ ဖြစ်ပါတယ်။ နောက်တစ်ချက်ကတော့ View Detail Link ထည့်ထားပြီး href အတွက် လိပ်စာကို url() Function နဲ့ ပေးထားပါတယ်။ ဒါကြောင့် အဲ့ဒီ Link ကို နှိပ်လိုက်ရင် /articles/detail/{id} Route ကို ရောက်သွားမှာဖြစ်ပါတယ် (Double Quote String ကိုသုံးထားတာ သတိပြုပါ)။ စမ်းကြည့်လိုက်ရင် ရလဒ်က ဒီလိုဖြစ်မှာပါ။



Bootstrap ကိုအသုံးပြုထားတာဖြစ်လို့ ဒီထက်ပိုသပ်ရပ်အောင် လုပ်မယ်ဆိုရင်လည်း အလွယ်တစ်ကူ လုပ်လို့ရနိုင်ပါတယ်။ ဒါပေမယ့် Bootstrap အကြောင်းကို ဒီနေရာမှာ အသေးစိတ် ထည့်မပြောနိုင်လို့ မလေ့လာဖူးတဲ့သူတွေအတွက် မျက်စိအရမ်းမရှုပ်အောင် ဒီလောက်ပဲ ထားပါမယ်။

လက်စနဲ့ အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တစ်ချို့ထည့်ကြည့်ပါဦးမယ်။ `ArticleController.php` ထဲက `index()` Method ကို အခုလိုပြင်ကြည့်လိုက်ပါ။

```
public function index()
{
    $data = Article::latest()->paginate(5);

    return view('articles.index', [
        'articles' => $data
    ]);
}
```

မူလက `Article::all()` ကို အသုံးပြုထားရာကနေ `Article::latest()` ကို ပြောင်းသုံးလိုက်တာပါ။ အဓိပ္ပါယ်က Record တွေကို ထုတ်ယူတဲ့အခါ နောက်ဆုံးထည့်သွင်းထားတဲ့ Record ကို အရင်

ထုတ်ယူမယ်၊ တနည်းအားဖြင့် ပြောင်းပြန်စီပြီး ထုတ်ယူမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ ပြီးတော့ `paginate()` Method ကိုလည်း သုံးထားပါသေးတယ်။ စာမျက်နှာတွေ ခွဲပြမယ်၊ တစ်မျက်နှာကို (၅) ခုပဲပြမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ ဒါတွေကို ကိုယ့်ဘာသာရေးရမယ်ဆိုရင် အလုပ်ရှုပ်ပါတယ်။ အခုတော့ Framework မှာ ဒီလို လုပ်ဆောင်ချက်တွေ အသင့်ပါလို့ ယူသုံးလိုက်ယုံပါပဲ။

စမ်းကြည့်နိုင်ဖို့အတွက် `articles/index` Template မှာ ဒီလိုလေးထပ်ထည့်ပေးပါ။

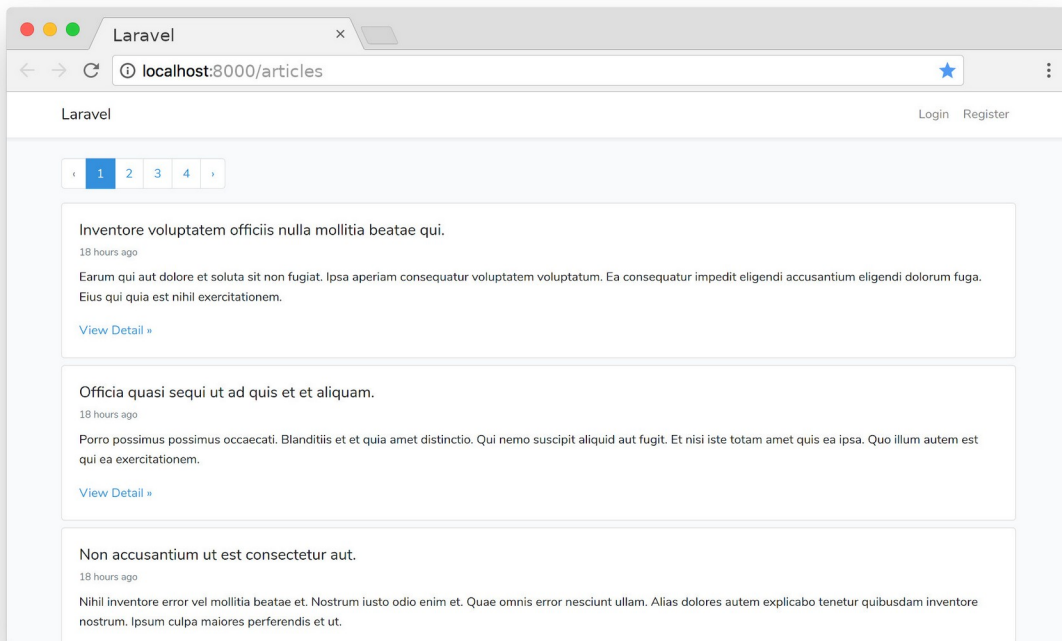
```
@extends("layouts.app")

@section("content")
    <div class="container">

        {{ $articles->links() }}

        @foreach($articles as $article)
            ...
        @endforeach
    </div>
@endsection
```

ကုန်တိုသွားအောင် အကုန်ပြန်မပြတော့ပါဘူး။ ထပ်တိုးလိုက်တဲ့အပိုင်းကိုပဲ ဦးစားပေးပြီး ပြထားတာပါ။ `$articles` ပေါ်မှာ `links()` Method ကို Run ထားတဲ့ ကုန်တစ်ကြောင်းပဲ ထပ်ဖြည့်လိုက်တာပါ။ ဒီ Method က စာမျက်နှာ ခွဲပြတဲ့အခါ နောက်စာမျက်နှာတွေကို သွားလို့ရတဲ့ Pagination Links တွေကို ထုတ်ပေးတဲ့ Method ပါ။ အတူတူပါပဲ။ ကိုယ်တိုင်ရေးရမယ်ဆိုရင် အလုပ်ရှုပ်နိုင်ပေမယ့် အခုတော့ Framework က ပေးထားတဲ့လုပ်ဆောင်ချက်ကို ယူသုံးလိုက်ယုံပါပဲ။ ရလဒ်က ဒီလိုဖြစ်သွားပါလိမ့်မယ်။



စာမျက်နှာ 1, 2, 3 စသဖြင့် Paging Links တွေ ပါဝင်သွားတာပါ။ နောက်တစ်ဆင့်အနေနဲ့ ArticleController ရဲ့ detail() Method ကို အခုလိုပြင်ပေးရပါမယ်။

```
public function detail($id)
{
    $data = Article::find($id);

    return view('articles.detail', [
        'article' => $data
    ]);
}
```

Article::find() Method ကိုသုံးပြီး ပေးလိုက်တဲ့ ID နဲ့ ကိုက်ညီတဲ့ Record တစ်ကြောင်းကို ထုတ်ယူလိုက်တာပါ။ ထုတ်ယူရရှိလာတဲ့ Data ကို သုံးပြီး Detail Template ကို ပြခိုင်းထားခြင်း ဖြစ်ပါတယ်။ ဒါကြောင့် Detail Template ထပ်ရေးပါမယ်။ /resources/views/articles ဖိုဒါထဲမှာ detail.blade.php အမည်နဲ့ View ဖိုင်တစ်ခု ထပ်တည်ဆောက်ပေးပါ။ ပြီးရင် ပေးထားတဲ့ကုဒ်ကို ရေးသားရမှာ ဖြစ်ပါတယ်။

```

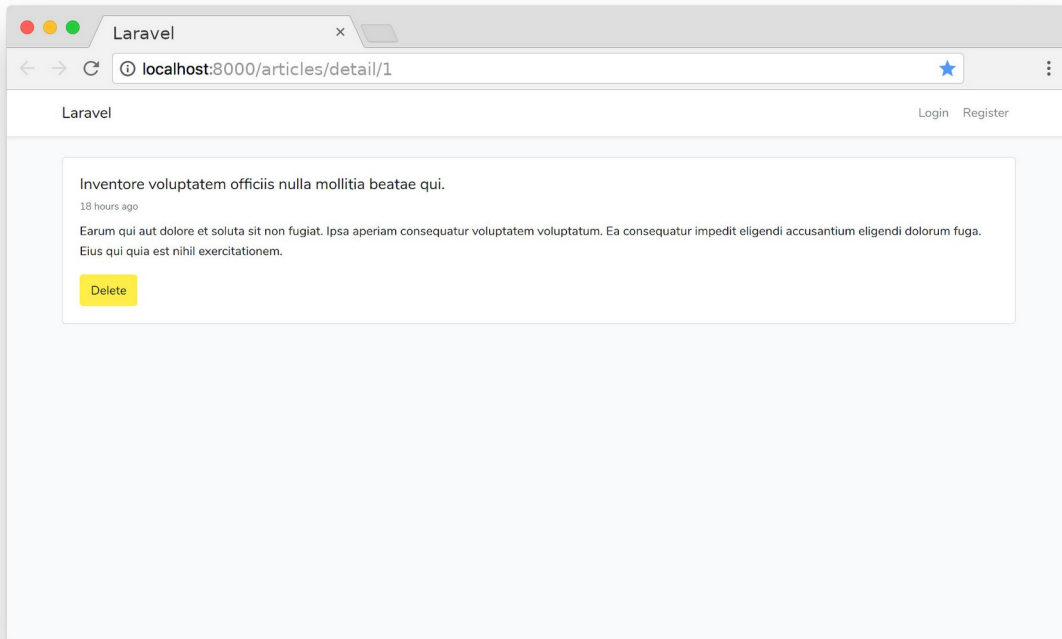
@extends("layouts.app")

@section("content")
    <div class="container">
        <div class="card mb-2">
            <div class="card-body">
                <h5 class="card-title">{{ $article->title }}</h5>
                <div class="card-subtitle mb-2 text-muted small">
                    {{ $article->created_at->diffForHumans() }}
                </div>
                <p class="card-text">{{ $article->body }}</p>
                <a class="btn btn-warning"
                    href="{{ url('/articles/delete/' . $article->id) }}">
                    Delete
                </a>
            </div>
        </div>
    </div>
@endsection

```

ဒီကုဒ်က Index Template အတွက်ကုဒ်နဲ့ သဘောသဘာဝ အတူတူပါပဲ။ ကွာသွားတာက Record တစ်ကြောင်းတည်းမို့လို့ `foreach()` တွေ့တာတွေနဲ့ Loop လုပ်နေစရာမလိုတော့ဘဲ ဖော်ပြထားခြင်း ဖြစ်ပါတယ်။ `$article` နဲ့ `$articles` မမှားပါစေနဲ့။ Variable အမည်လေးတွေ ဂရုစိုက်ပါ။ အရမ်းမှားတတ်ကြပါတယ်။ ဒီ Template ရဲ့ နောက်ထပ်ထူးခြားချက်ကတော့ View Detail မပါတော့ဘဲ Delete ခလုပ်တစ်ခု ပါဝင်သွားခြင်း ဖြစ်ပါတယ်။ အဲ့ဒီ Delete ခလုပ်ကို နှိပ်လိုက်ရင် `/articles/delete/{id}` Route ကို သွားမှာပါ (Double Quote String ကိုသုံးတာ သတိပြုပါ။ မှားကြလွန်းလို့ သတိပေးရတာပါ)။

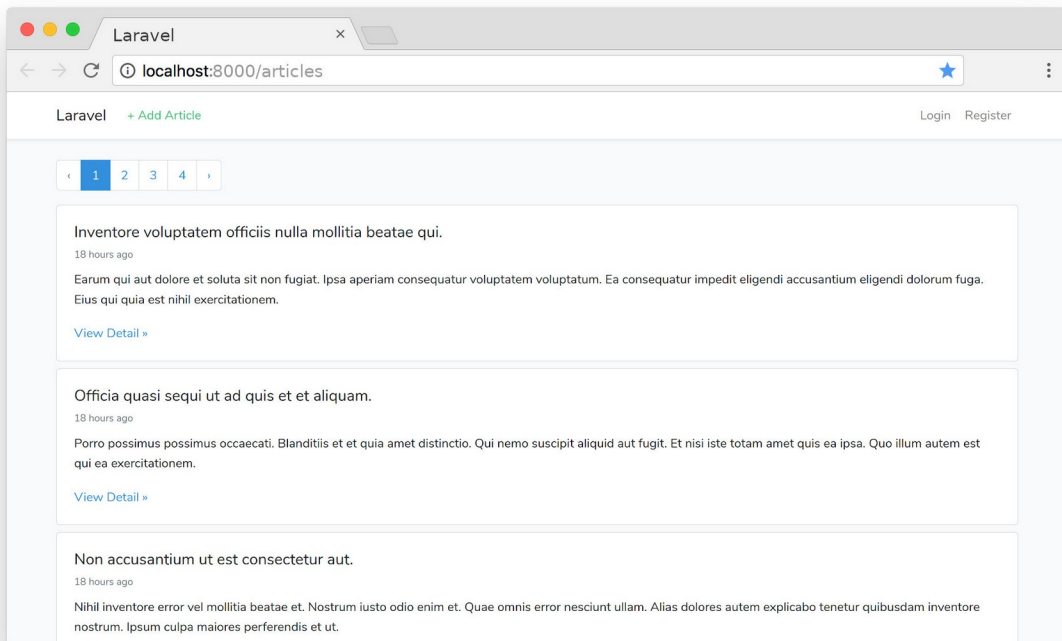
`/articles/delete/{id}` Route မရှိသေးပါဘူး။ ရေးပေးရမှာပါ။ အခုမရေးသေးပါဘူး။ နောက်မှ ရေးပါမယ်။ လက်ရှိအနေအထားကို စမ်းကြည့်လို့ရပါပြီ။ Article List ထဲက နှစ်သက်ရာတစ်ခုကို View Detail နှိပ်ကြည့်ရင် အခုလိုရလဒ်ကို ရရှိပါလိမ့်မယ်။



ဆက်လက်ပြီး ဒီအခန်းအတွက် နောက်ဆုံး Template နဲ့ပတ်သက်တဲ့ ဖြည့်စွက်ချက်အနေနဲ့ `/resources/views/layouts` ထဲက `app.blade.php` ကိုဖွင့်ကြည့်ပါ။ `<!-- Left Side of Navbar -->` ဆိုတဲ့နေရာလေးကို ရှာပြီး အခုလိုဖြည့်စွက်ပေးလိုက်ပါ။

```
<!-- Left Side Of Navbar -->
<ul class="navbar-nav mr-auto">
  <li class="nav-item">
    <a class="nav-link text-success"
      href="{{ url('/articles/add') }}">
      + Add Article
    </a>
  </li>
</ul>
```

ဒါဟာ Article အသစ်တွေ ထပ်ထည့်နိုင်ဖို့အတွက် Add Article ခလုပ်ကို ထည့်သွင်းလိုက်ခြင်းဖြစ်ပါတယ်။ နှိပ်လိုက်ရင် `/articles/add` Route ကို ရောက်သွားမှာပါ။ အဲ့ဒီ Route လည်း မရေးရသေးပါဘူး။ နောက်တစ်ခန်းကျတော့မှ ဆက်ရေးမှာ ဖြစ်ပါတယ်။ အခုနေ ရလဒ်ကတော့ ဒီလိုဖြစ်မှာပါ။



Laravel ဆိုတဲ့ ပရောဂျက်ခေါင်းစဉ် ဘေးနားမှာ **+ Add Article** ခလုပ်တစ်ခု ဝင်သွားတာပါ။ အလုပ်တော့ မလုပ်သေးပါဘူး။ အလုပ်လုပ်အောင် နောက်တစ်ခန်းမှာ ဆက်ရေးကြမှာ ဖြစ်ပါတယ်။

ဖြည့်စွက်မှတ်သားသင့်တာကတော့၊ Laravel ဆိုတဲ့ ပရောဂျက်ခေါင်းစဉ်ကို ပြင်ချင်ရင် `.env` ဖိုင်ထဲက `APP_NAME` ကို ကိုယ်ကြိုက်တဲ့ နာမည်နဲ့ ပြင်ပေးလိုက်လို့ ရတယ်ဆိုတဲ့အချက်ပဲ ဖြစ်ပါတယ်။ ဥပမာ -

```
APP_NAME="Laravel Blog"
```

အခန်း (၁၁) - Form

ဒီအခန်းမှာတော့ HTML Form နဲ့ Request Data တွေကို စီမံတဲ့အပိုင်း ဆက်ကြပါမယ်။ UI ပိုင်းက လိုအပ်မယ့် ခလုပ်တွေတော့ ထည့်ခဲ့ပြီးသားပါ။ လိုအပ်မယ့် Route တွေ မထည့်ရသေးလို့ ထပ်ထည့်ရပါဦးမယ်။ ဒီလိုပါ -

```
Route::get('/articles/add', 'ArticleController@add');  
Route::post('/articles/add', 'ArticleController@create');  
Route::get('/articles/delete/{id}', 'ArticleController@delete');
```

Add အတွက် Route နှစ်ခုနဲ့ Delete အတွက် တစ်ခုပါ။ Add အတွက် Route က နှစ်ခုဆိုပေမယ့် URL လိပ်စာက တစ်ခုတည်းပါ။ သတိထားကြည့်ပါ။ တူညီတဲ့လိပ်စာကိုပဲ `get()` Method နဲ့တစ်ခု၊ `post()` Method နဲ့တစ်ခု ရေးထားတာပါ။ လိပ်စာရိုက်ထည့်ပြီး (သို့) Link ခလုပ်ကိုနှိပ်ပြီး `/articles/add` ကို လာရင် `ArticleController@add` အလုပ်လုပ်သွားမှာဖြစ်ပြီး၊ HTML Form ကနေ Submit ခလုပ်နှိပ်ပြီး `/articles/add` ကို လာရင်တော့ `ArticleController@create` အလုပ်လုပ်သွားမှာပါ။

ဆက်လက်ပြီး HTML Form ပါဝင်တဲ့ View Template တစ်ခု တည်ဆောက်ပါမယ်။ `/resources/views/articles` ဖိုဒါထဲမှာ `add.blade.php` အမည်နဲ့ ဖိုင်တစ်ခုဆောက်ပါ။ ပြီးရင် ဒီ HTML Form ကုဒ်ကို ရေးသားပေးပါ။

```

@extends('layouts.app')

@section('content')
    <div class="container">
        <form method="post">
            @csrf
            <div class="form-group">
                <label>Title</label>
                <input type="text" name="title" class="form-control">
            </div>
            <div class="form-group">
                <label>Body</label>
                <textarea name="body" class="form-control"></textarea>
            </div>
            <div class="form-group">
                <label>Category</label>
                <select class="form-control" name="category_id">
                    @foreach($categories as $category)
                        <option value="{{ $category['id'] }}">
                            {{ $category['name'] }}
                        </option>
                    @endforeach
                </select>
            </div>
            <input type="submit" value="Add Article"
                class="btn btn-primary">
        </form>
    </div>
@endsection

```

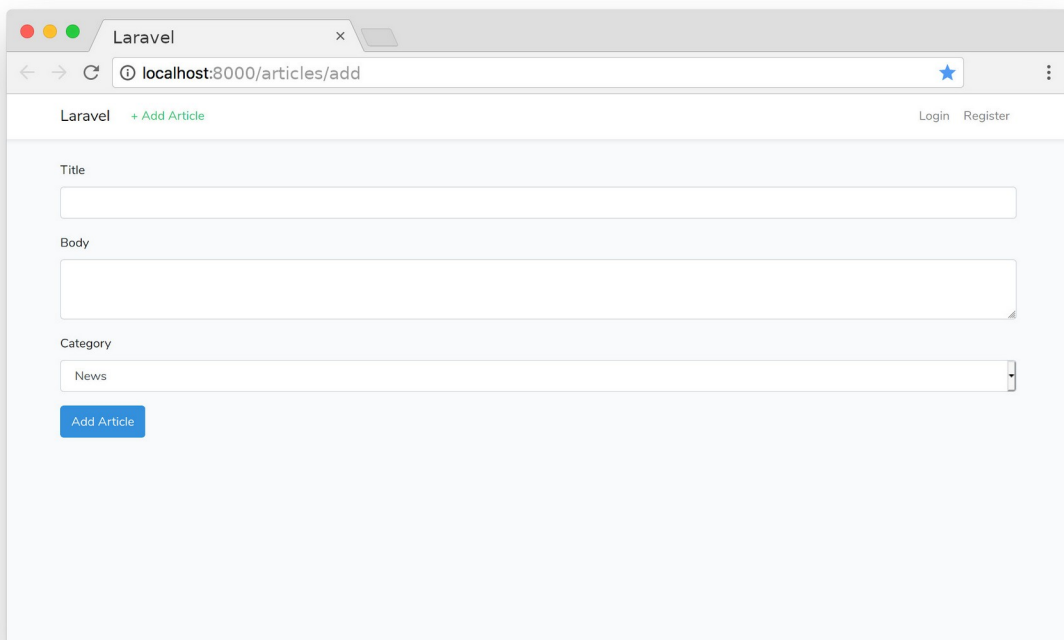
ဒီကုဒ်မှာ အထူးသဖြင့် သတိပြုသင့်တာ (၂) ချက်ရှိပါတယ်။ ပထမတစ်ချက်ကတော့ <form> Element အတွင်းမှာ @csrf လို့ခေါ်တဲ့ Blade Directive တစ်ခု ပါဝင်ပါတယ်။ Cross-site Request Forgery လို့ ခေါ်တဲ့ လုံခြုံရေးပြဿတစ်ခုရှိပြီး Laravel က ဒီပြဿနာကို ဖြေရှင်းပေးထားခြင်း ဖြစ်ပါတယ်။ CSRF အကြောင်းကိုတော့ ကြားဖြတ် မပြောနိုင်ပါဘူး။ လိုရင်းအနေနဲ့ လုံခြုံရေးအတွက် လိုအပ်တဲ့ လုပ်ဆောင် ချက်ဖြစ်ပြီး @csrf မပါရင် Laravel က အဲဒီဖောင်ကပေးပို့တဲ့ လုပ်ငန်းကို လက်ခံ အလုပ်လုပ်မှာ မဟုတ် ဘူးလို့ပဲ မှတ်ထားပေးပါ။ ဒုတိယတစ်ချက်ကတော့ category_id Select Box အတွက် \$categories ကို Loop လုံးပြီး ရေးသားထားလို့ ဒီ View Template အလုပ်လုပ်ဖို့အတွက် \$categories လိုအပ်မှာဖြစ်ပါတယ်။ ကျန်တာတွေကတော့ ရိုးရိုး HTML Form တစ်ခုမှာ ရေးလေ့ရေး ထ ရှိတဲ့ ကုဒ်တွေချည်းပါပဲ။

အလုပ်တော့ လုပ်ဦးမှာ မဟုတ်ပါဘူး။ `ArticleController@add` Method ကို မရေးရသေးတဲ့ အတွက်ပါ။ ဒါကြောင့် `ArticleController` ထဲမှာ `add()` Method ကို အခုလိုရေးပေးပါ။

```
public function add()
{
    $data = [
        [ "id" => 1, "name" => "News" ],
        [ "id" => 2, "name" => "Tech" ],
    ];

    return view('articles.add', [
        'categories' => $data
    ]);
}
```

`articles/add` Template ကို `$categories` နဲ့အတူ ပြခိုင်းတဲ့ကုန်ဖြစ်ပါတယ်။ ဒါကြောင့်အခုနေ စမ်းကြည့်မယ်ဆိုရင် စမ်းကြည့်လို့ရပါပြီ။ `/articles/add` လို့ URL လိပ်စာ ရိုက်ထည့်လို့ရသလို ပြီးခဲ့တဲ့အခန်းမှာ ထည့်ခဲ့တဲ့ **+ Add Article** ခလုပ်ကို နှိပ်လို့လည်း ရပါတယ်။



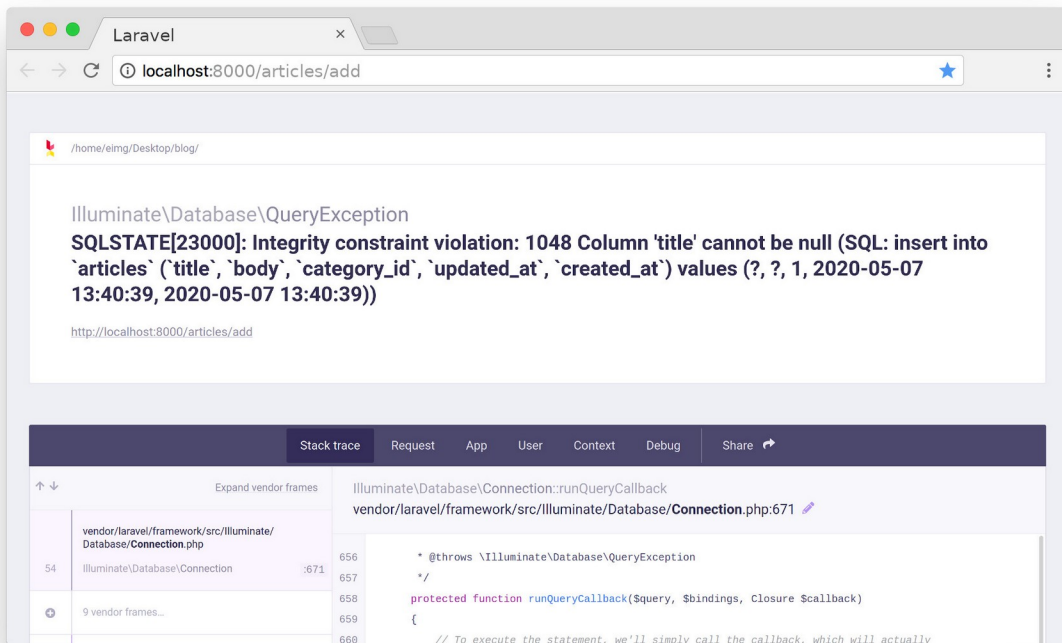
Add Article ဖောင်ကိုတော့ မြင်ရပါပြီ။ နမူနာစမ်းထည့်လို့တော့ မရသေးပါဘူး။ ဒီဖောင်ကနေ **Add Article** ခလပ်ကို နှိပ်လိုက်ရင် `/articles/add` Route ကို POST နဲ့သွားမှာဖြစ်ပါတယ်။ ဒါကြောင့် `ArticleController@create` Method ရေးပေးဖို့ လိုပါတယ်။ `create()` Method ရဲ့ တာဝန်ကတော့ ဖောင်မှာ ရေးဖြည့်လိုက်တဲ့ အချက်အလက်တွေကို သိမ်းပေးရမှာပါ။ ဒီလိုရေးရပါတယ်။

```
public function create()
{
    $article = new Article;
    $article->title = request()->title;
    $article->body = request()->body;
    $article->category_id = request()->category_id;
    $article->save();

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

ပထမဆုံး `Article Model` အသစ်တစ်ခုတည်ဆောက်ပြီး `title`, `body`, `category_id` ဆိုတဲ့ Property တွေ သတ်မှတ်လိုက်တာပါ။ ဖောင်ကပေးပို့တဲ့ Data ကို လိုချင်ရင် `Request $request` Object (သို့မဟုတ်) `request()` Function ကနေယူလို့ ရပါတယ်။ နမူနာမှာတော့ `request()` Function ကနေ ယူထားပါတယ်။ လိုအပ်တဲ့ Property တွေသတ်မှတ်ပြီး `save()` ကို Run ပေးလိုက်ရင် ရပါပြီ။ `INSERT INTO` တွေဘာတွေ ကိုယ့်ဘာသာ လုပ်နေစရာမလိုဘဲ Laravel က Table ထဲမှာ Record အသစ်တစ်ခုကို ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်။ စမ်းကြည့်လို့ရပါပြီ။

စမ်းကြည့်တဲ့အခါ ဖောင်မှာ Data ကို စုံအောင်မဖြည့်ပဲ စမ်းခဲ့ရင် ဒီလို Error တက်နိုင်ပါတယ်။



လိုအပ်တဲ့ Data မစုံဘဲ INSERT လုပ်ဖို့ ကြိုးစားမိသလို ဖြစ်သွားလို့ အခုလို Error တက်တာပါ။ ဒါမျိုး Error မတက်စေဖို့အတွက် Validation စစ်ပေးသင့်ပါတယ်။ ဒါကြောင့် `create()` Method ကို အခုလို ပြင်လိုက်ပါ။

```
public function create()
{
    $validator = validator(request()->all(), [
        'title' => 'required',
        'body' => 'required',
        'category_id' => 'required',
    ]);

    if($validator->fails()) {
        return back()->withErrors($validator);
    }

    $article = new Article;
    $article->title = request()->title;
    $article->body = request()->body;
    $article->category_id = request()->category_id;
    $article->save();

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

Laravel နဲ့အတူ တစ်ခါတည်းပါတဲ့ `validator()` Function ကိုသုံးပြီး Validation စစ်နိုင်ပါတယ်။ Parameter နှစ်ခုပေးရပြီး ပထမတစ်ခုက Data ပါ။ နမူနာမှာ `request()->all()` လို့ပြောထားတဲ့ အတွက် Form Request Data အားလုံးကို စစ်မှာပါ။ ဒုတိယက Validation Rule ဖြစ်ပါတယ်။ `required, email, minlength` စသဖြင့် လိုအပ်တဲ့ Rule တွေ သတ်မှတ်နိုင်ပါတယ်။ နမူနာမှာ တော့ `title, body, category_id` အားလုံးအတွက် `required` လို့ပြောထားလို့ မဖြစ်မနေပေါ့ရ မယ်လို့ စစ်လိုက်တာပါ။ တစ်ခြားသုံးလိုရတဲ့ Validation Rule တွေကို သိချင်ရင် အောက်ကလင့်မှာ ကြည့် လို့ရပါတယ်။

- <https://laravel.com/docs/validation#available-validation-rules>

ပြီးတဲ့အခါ `fails()` Method နဲ့စစ်လိုက်ပြီး Validation Fails ဖြစ်ခဲ့ရင် `return back()` လို့ပြော လိုက်တဲ့အတွက် အောက်က အလုပ်တွေ ဆက်မလုပ်တော့ဘဲ လာခဲ့တဲ့နေရာကို ပြန်ရောက်သွားမှာ ဖြစ်ပါ တယ်။ ဒါကြောင့် Error မတက်တော့ဘဲ Add Form ကို ပြန်ရောက်သွားမှာပါ။ ဒီလိုသွားတဲ့အခါ `withErrors()` ရဲ့ အကူအညီနဲ့ `$validator` Object ကို သယ်သွားလို့ Form Template မှာ အဲ့ဒီ `$validator` Object ထဲက Error Message တွေကို ပြချင်ရင်ပြလို့ရပါတယ်။

အခုနေစမ်းကြည့်ရင် Error မတက်တော့ဘဲ Form Template ကို ပြန်ရမှာပါ။ ဘာ Error Message မှ လည်း ပြမှာ မဟုတ်ပါဘူး။ ဒါကြောင့် ပြသင့်တဲ့ Error Message တွေ ပြစေဖို့အတွက် add Template မှာ အခုလို ဖြည့်စွက်ပေးဖို့ လိုအပ်ပါတယ်။

```

@extends('layouts.app')

@section('content')
    <div class="container">

        @if($errors->any())
            <div class="alert alert-warning">
                <ol>
                    @foreach($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ol>
            </div>
        @endif

        <form method="post">
            ...
        </form>
    </div>
@endsection

```

Template ကုဒ်ထဲမှာ `$errors` Variable ကို သုံးလို့ ရနေတာကို အရင်သတိပြုပါ။ `withErrors()` နဲ့ Controller ကပေးလိုက်လို့ အခုလိုသုံးလို့ရနေတာပါ (နောက်က `s` ကလေးတွေကို ဂရုစိုက်ပါ။ အရမ်း ကျန်ကြပါတယ်။)။ `any()` Method နဲ့ ရှိမရှိအရင်စစ်ပြီး ရှိရင် `all()` Method နဲ့ ရှိသမျှ Error အားလုံး ကို Loop ပါတ်ပြီး ဖော်ပြလိုက်တာ ဖြစ်ပါတယ်။ ဒါကြောင့် အခုနေ ဖောင်မှာ စုံအောင်မဖြည့်ဘဲ ခလုပ်နှိပ် ရင် အခုလို Error Message ကို ရရှိမှာပဲ ဖြစ်ပါတယ်။

အခုဆိုရင် Article အသစ်တွေ ထည့်လို့ရသွားပါပြီ။ Validation လည်း စစ်ပြီးပါပြီ။ လက်စနဲ့ ထည့်ထားတဲ့ Article တွေကို ပြန်ဖျက်လို့ရတဲ့ လုပ်ဆောင်ချက်ကို ထပ်ထည့်ပါမယ်။ Route တွေ ခလုပ်တွေက ထည့်ပြီး သားပါ။ ArticleController မှာ `delete()` Method ကို အခုလို ထပ်ရေးပေးဖို့ပဲ လိုပါတယ်။

```
public function delete($id)
{
    $article = Article::find($id);
    $article->delete();

    return redirect('/articles')->with('info', 'Article deleted');
}
```

ဒါပါပဲ။ လွယ်ပါတယ်။ `find()` Method ကိုသုံးပြီး ID နဲ့ ကိုက်တဲ့ Article ကိုထုတ်ယူပါတယ်။ ပြီးတဲ့အခါ သူ့ပေါ်မှာ `delete()` ကို Run ပေးလိုက်ယုံပါပဲ။ စမ်းကြည့်နိုင်ဖို့အတွက် Article Detail ကိုသွားပြီး Delete ခလုပ်ကို နှိပ်ရမှာပါ။ ဖျက်ပြီးသွားရင် Article List ကို ပြန်သွားခိုင်းထားပါတယ်။ အဲ့ဒီလို သွားခိုင်း တဲ့အခါမှာ `with()` Method နဲ့ အချက်အလက်တစ်ချို့ ပေးလိုက်တာကို သတိပြုပါ။ အဲ့ဒါကို Flash Message လို့ ခေါ်ပါတယ်။ User သိဖို့လိုတဲ့ အချက်အလက်တွေကို တစ်ကြိမ် ပြပေးတဲ့ Message အမျိုး အစားပါ။ အခုတော့ ပြဦးမှာ မဟုတ်သေးပါဘူး။ ပြတဲ့ကုဒ်ကို ရေးပေးရဦးမှာပါ။ ဒါကြောင့်

articles/index.blade.php ဖိုင်မှာ အခုလို ဖြည့်စွက်ပေးရပါဦးမယ်။

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

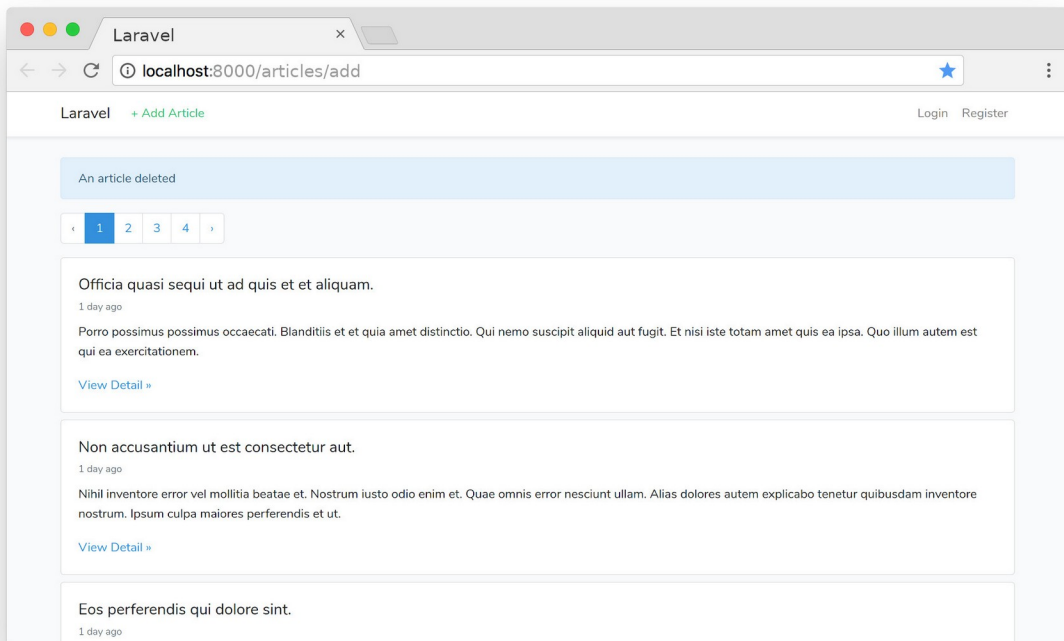
@section("content")
    <div class="container">

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

        {{ $articles->links() }}

        @foreach($articles as $article)
            ...
        @endforeach
    </div>
@endsection
```

with() နဲ့ပေးလိုက်တဲ့ Flash Message တွေက session() ထဲမှာ သိမ်းထားတာဖြစ်လို့ အရင်ဆုံး ရှိမရှိ စစ်ပါတယ်။ ရှိတယ်ဆိုတော့မှ ပြခိုင်းလိုက်တာပါ။ ဒါကြောင့် အခုနေ Article တစ်ခုကို Delete လုပ်လိုက်ရင် ရလဒ်က အခုလိုဖြစ်မှာပါ။



အခုဆိုရင် Data တွေရယူပုံ၊ သိမ်းဆည်းပုံ၊ Delete လုပ်ပုံ ဒါတွေအားလုံး စုံသွားပါပြီ။ Edit ပြုလုပ်ပုံကို တော့ နမူနာ ထည့်မပေးတော့ပါဘူး။ အခုပေးထားသလောက်ကိုပဲ ကောင်းကောင်း သဘောပေါက်အောင် လေ့ကျင့်ထားပါ။ အလားတူလုပ်ဆောင်ချက်မျိုးတွေကို ကြည့်စရာမလိုတော့ဘဲ လက်တမ်း ရေးနိုင်တဲ့ အဆင့်ထိ လုပ်ထားဖို့လိုပါတယ်။ ဒီလိုလုပ်ထားလို့ MVC ကုန်တွေနဲ့ Laravel ရဲ့ သဘောသဘာဝကို နားလည်နေပြီဆိုရင် Edit လို လုပ်ဆောင်ချက်မျိုးကို ကိုယ်တိုင်ရေးဖြည့်လို့ ရသွားပါတယ်။ ရေးရတာမ ခက်ပါဘူး၊ ရေးရတဲ့ကုန်များမှာမို့လို့သာ အစပိုင်းမှာ ခေါင်းမူးသွားမှာစိုးလို့ ချန်ထားခဲ့တာပါ။

အခန်း (၁၂) - Relationship

Database Table ထဲက Data တွေကို စီမံတဲ့အခါ ရိုးရိုး SQL Query တွေနဲ့ဆိုရင် Table Relationship တွေအတွက် JOIN Query တွေကို သုံးကြရပါတယ်။ One to One, One to Many, Many to Many စသဖြင့် Relationship ပုံစံအမျိုးမျိုး ရှိတဲ့ထဲက အသုံးအများဆုံးဖြစ်တဲ့ One to Many Relationship ကို Laravel မှာ ဘယ်လိုစီမံရသလဲ ဆိုတာကို ဆက်လက် လေ့လာသွားကြမှာ ဖြစ်ပါတယ်။

ဥပမာ - စာသင်ခန်းတစ်ခန်းမှာ ကျောင်းသားတွေ အများကြီး ရှိတယ်ဆိုပါစို့။ ဒါဟာ One to Many Relationship ပုံစံဆက်စပ်မှုပါ။ ဒီအတွက် Laravel မှာ Relationship Method နှစ်ခု မှတ်သင့်ပါတယ်။ `hasMany()` ဆိုတဲ့ Method နဲ့ `belongsTo()` ဆိုတဲ့ Method ပါ (s သတိထားပါ)။ စာသင်ခန်းတစ်ခန်းမှာ ကျောင်းသားတွေ အများကြီး ရှိတယ်ဆိုတာကို A classroom has many students လို့ ပြောနိုင်ပါတယ်။ ဒါကြောင့် Classroom Model တစ်ခုပေါ်မှာ `hasMany('Student')` ပြောလိုက်ရင် အဲ့ဒီ Classroom နဲ့သက်ဆိုင်တဲ့ ကျောင်းသားစာရင်းကို ပြန်ရနိုင်ပါတယ်။

ကျောင်းသား ဘိုဘို ဟာ Classroom A မှာ တက်နေတယ်ဆိုရင် Bobo belongs to Classroom A လို့ ပြောနိုင်ပါတယ်။ ဒါကြောင့် Bobo ဆိုတဲ့ Student Model ပေါ်မှာ `belongsTo('Classroom')` လို့ ပြောလိုက်ရင် Classroom A ကို ပြန်ရမှာပါ။ နားလည်မယ်လို့ ယူဆပါတယ်။

ဒီလို အလုပ်လုပ်နိုင်ဖို့အတွက် Table Name တွေ Column Name တွေ ပေးပုံပေးနည်း မှန်ဖို့တော့ လိုပါတယ်။ မမှန်လည်း ရပေမယ့် အမည်ပေးပုံမှန်မှသာ Setting တွေ ပြင်စရာမလိုဘဲ ဒီလုပ်ဆောင်ချက်ကို အလိုအလျောက်တန်းရမှာပါ။ Convention over Configuration အကြောင်းပြောခဲ့တာ မှတ်မိကြဦးမှာပါ။

အများကြီးမှတ်စရာမလိုပါဘူး။ `id` နဲ့ `_id` ချိတ်တယ်လို့ မှတ်ထားလိုက်ရင်ရပါပြီ။ ဆိုလိုတာက Classroom A ရဲ့ `id` က 12 ဆိုရင် ကျောင်းသား Bobo ရဲ့ `classroom_id` တန်ဖိုးကို 12 လို့ပေးလိုက်ရင်ရပါပြီ။ တစ်ခြားဘာမှ မလိုအပ်ပါဘူး။ `Bobo->belongsTo('Classroom')` လို့ပြောလိုက်တဲ့အခါ Laravel ရဲ့ Bobo Model ရဲ့ `classroom_id` ကို ကြည့်လိုက်မှာပါ။ `classroom_id` တန်ဖိုး 12 ဖြစ်နေတဲ့အခါ `classrooms` Table ရဲ့ `id` တန်ဖိုး 12 နဲ့ ကိုက်ညီတဲ့ Record ကို ပြန်ပေးသွားမှာပါ။ မျက်စိရှုပ်သွားရင် ဒီစာပိုဒ်ကို နောက်ထပ် တစ်ခေါက်နှစ်ခေါက်လောက် ပြန်ဖတ်ကြည့်ပေးပါ။ မျက်စိထဲမှာ ဒီဆက်စပ်မှုကို ရှင်းနေအောင်မြင်မှ ရှေ့ဆက်သွားလို့ ကောင်းမှာပါ။

အလားတူပဲ `ClassroomA->hasMany('Student')` လို့ပြောလိုက်ရင် ပြောလိုက်ရင် Classroom A ရဲ့ `id` ကို ယူမှာပါ။ `id` တန်ဖိုးက 12 ဖြစ်နေတဲ့အခါ `students` Table ထဲက `classroom_id` တန်ဖိုး 12 ဖြစ်နေသူတွေ စာရင်းကို ပြန်ပေးလိုက်မှာ ဖြစ်ပါတယ်။ ဒီအလုပ်တွေကို Framework က အကုန်လုပ်ပေးသွားပြီး ကိုယ့်ဘက်က `id` နဲ့ `_id` ချိတ်ပြီး အလုပ်လုပ်ပေးတယ်ဆိုတာကို မှတ်ထားရင် ရပါပြီ။

လက်ရှိရေးလက်စကုဒ်မှာ ထည့်သွင်းစမ်းသပ်နိုင်ဖို့အတွက် Table (၂) ခု ထပ်မံ တည်ဆောက်ပါမယ်။ `categories` Table နဲ့ `comments` Table တို့ပါ။ ဒါကြောင့် Model ဖိုင်တွေ Migration ဖိုင်တွေ အခုလို တည်ဆောက်လိုက်ပါမယ်။

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

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

ပြီးတဲ့အခါ Table ရဲ့ ဖွဲ့စည်းပုံသတ်မှတ်ဖို့အတွက် Migration ဖိုင်တွေကို ပြင်ပါမယ်။ `/databases/migrations` ဖိုဒါထဲက `xxx_create_categories_table.php` ဖိုင်ကိုဖွင့်ပါ။ `up()` Method ကို အခုလို ပြင်ပေးပါ။

```
public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->timestamps();
    });
}
```

တစ်ကြောင်းတည်း ထပ်တိုးလိုက်တာပါ။ name အမည်နဲ့ String Column တစ်ခုဖြစ်ပါတယ်။ ပြီးတဲ့အခါ xxx_create_comments_table.php ဖိုင်ကိုဖွင့်ပြီး up() Method ကို အခုလိုပြင်ပေးပါ။

```
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->id();
        $table->text('content');
        $table->integer('article_id');
        $table->timestamps();
    });
}
```

comments Table မှာတော့ Column (၂) ခုထပ်တိုးထားပါတယ်။ content နဲ့ article_id တို့ပါ။ ဒါတွေ သတ်မှတ်ပြီးရင် migrate Command ကို အခုလို Run ပေးပါ။

```
php artisan migrate:fresh
```

migrate လို့ Run ရင် ထပ်တိုးလိုက်တဲ့ဖိုင်တွေကိုပဲ ရွေးပြီး Run သွားမှာပါ။ Laravel က သိပါတယ်၊ မှတ်ထားပါတယ်။ အခုတော့ migrate:fresh လို့ Run ထားတဲ့အတွက်ကြောင့် ရှိသမျှဖိုင်အားလုံးကို အစကနေ ပြန် Run သွားမှာပါ။ အရင်ကစမ်းထားတဲ့ Data တွေတစ်ခါတည်း ရှင်းပြီးသား ဖြစ်စေချင်လို့ အခုလို Run ထားတာပါ။

ပြီးတဲ့အခါ နမူနာ Data တွေထည့်နိုင်ဖို့အတွက် Model Factory တွေ Seed တွေ ရေးပါတယ်။ Model Factory ဖိုင်တွေကိုအခုလို တည်ဆောက်ပေးပါ။

```
php artisan make:factory CategoryFactory --model=Category
php artisan make:factory CommentFactory --model=Comment
```

နောက်ကနေ `--model` Option နဲ့အတူ Model အမည်တစ်ခါတည်း တွဲထည့်ပေးလိုက်တာကို သတိပြုပါ။ ဟိုး Migration အခန်းမှာတုန်းက Factory ကုန်တွေပြင်တဲ့အခါ Model အမည်ကို ကိုယ့်ဘာသာ ပြင်ပေးရပါတယ်။ အခု အဲဒီလို ကိုယ်ဘာသာ Model အမည်ကို ပြင်စရာမလိုအောင် ဖိုင် Generate လုပ်ကတည်းက တစ်ခါတည်း ပြောလိုက်တဲ့သဘောပါ။

/databases/factories/ ဖိုင်ထဲက CategoryFactory.php ကိုဖွင့်ပြီး အခုလိုပြင်ပေးပါ။

```
$factory->define(Category::class, function (Faker $faker) {
    return [
        "name" => ucwords($faker->word)
    ];
});
```

Faker နဲ့ Random Word တစ်ခုကို Category Name အဖြစ် သတ်မှတ်ပေးလိုက်တာပါ။ `ucwords()` ကတော့ အဲဒီ Word ကို Capital Case ဖြစ်စေချင်တဲ့အတွက် သုံးလိုက်တာပါ။ Standard PHP Function တစ်ခု ဖြစ်ပါတယ်။ ပြီးတဲ့အခါ CommentFactory.php မှာ အခုလိုပြင်ပေးပါ။

```
$factory->define(Comment::class, function (Faker $faker) {
    return [
        "content" => $faker->paragraph,
        "article_id" => rand(1, 20),
    ];
});
```

Comment Content အနေနဲ့ Random Paragraph တစ်ခုဖြစ်ပြီး `article_id` အတွက်တော့ 1, 20 ကြား Random တန်ဖိုးတစ်ခုကို သတ်မှတ်ပေးလိုက်တာပါ။ ပြီးတဲ့အခါ /databases/seeds ထဲက DatabaseSeeder.php မှာ အခုလိုပြင်ပေးပါ။

```
public function run()
{
    factory(App\Article::class, 20)->create();
    factory(App\Category::class, 5)->create();
    factory(App\Comment::class, 40)->create();
}
```

ဒါကြောင့် Run လိုက်ရင် Article အခု (၂၀)၊ Category (၅) ခု နဲ့ Comment အခု (၄၀) တို့ကို Sample Data အနေနဲ့ ရရှိမှာဖြစ်ပါတယ်။

```
php artisan db:seed
```

အခုဆိုရင် စမ်းဖို့အတွက် Table နဲ့ Data တွေတော့စုံသွားပါပြီ။ Relationship ကုန်တွေ စရေးပါတော့မယ်။ Article နဲ့ Category ရဲ့ ဆက်စပ်မှုက An Article belongs to a Category ဖြစ်ပါတယ်။ ဆိုလိုတာက Article တစ်ခုဟာ Category တစ်ခုနဲ့ သက်ဆိုင်ပါတယ်။ Article နဲ့ Comment ရဲ့ ဆက်စပ်မှုကတော့ An Article has many Comment ဖြစ်ပါတယ်။ Article တစ်ခုမှာ Comment တွေ အများကြီး ရှိတယ်ဆိုတဲ့ သဘောပါ။ ဒီသဘောတွေ ပေါ်လွင်အောင် ရေးမှာ ဖြစ်ပါတယ်။ /app ဖိုဒါထဲက Article.php ကိုဖွင့်ပါ။ ပြီးရင် အခုလိုရေးပေးပါ။

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    public function category()
    {
        return $this->belongsTo('App\Category');
    }

    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```


`category()` နဲ့ `comments()` ဆိုတဲ့ Method နှစ်ခုထပ်တိုးလိုက်တာပါ (ထပ်သတိပေးပါမယ်၊ `s` ကို ဂရုစိုက်ပါ)။ ဒီလိုရေးပေးလိုက်တဲ့အတွက် `category()` Method ကို ခေါ်ရင် လက်ရှိ Article နဲ့ သက်ဆိုင်တဲ့ Category ကို ရမှာပါ။ `comments()` Method ကို ခေါ်ရင်တော့ လက်ရှိ Article နဲ့ သက်ဆိုင်တဲ့ Comments တွေကို ရမှာပါ။

လက်တွေ့စမ်းသပ်နိုင်ဖို့ `/resources/views/articles` ထဲက `detail.blade.php` ကို အခု လိုပြင်ပေးပါ။

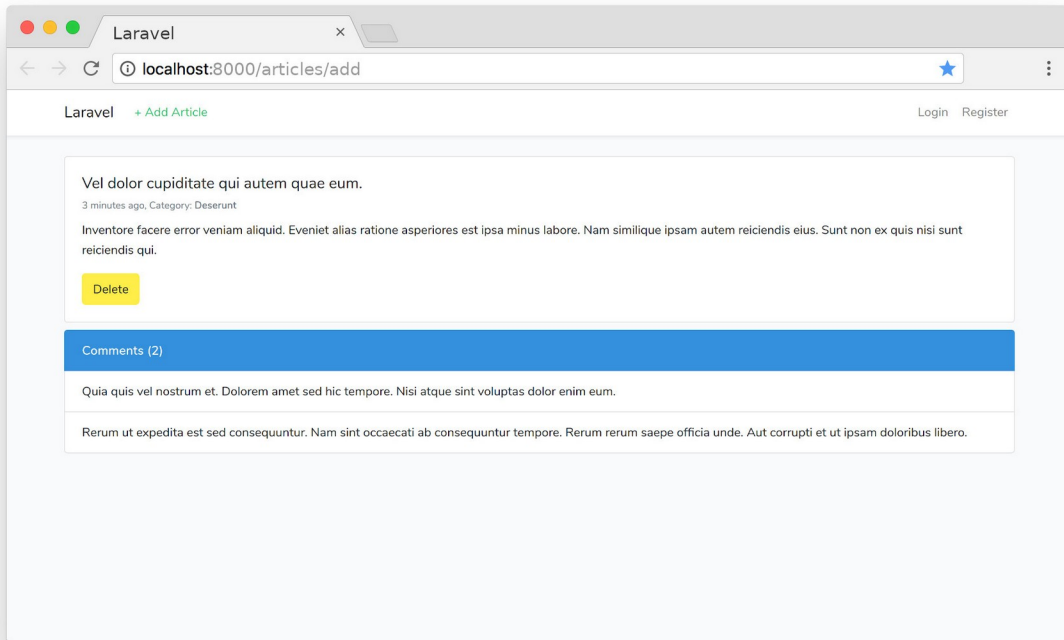
```
@extends("layouts.app")

@section("content")
    <div class="container">
        <div class="card mb-2">
            <div class="card-body">
                <h5 class="card-title">{{ $article->title }}</h5>
                <div class="card-subtitle mb-2 text-muted small">
                    {{ $article->created_at->diffForHumans() }},
                    Category: <b>{{ $article->category->name }}</b>
                </div>
                <p class="card-text">{{ $article->body }}</p>
                <a class="btn btn-warning"
                    href="{{ url("/articles/delete/$article->id") }}">
                    Delete
                </a>
            </div>
        </div>

        <ul class="list-group">
            <li class="list-group-item active">
                <b>Comments ({{ count($article->comments) }})</b>
            </li>
            @foreach($article->comments as $comment)
                <li class="list-group-item">
                    {{ $comment->content }}
                </li>
            @endforeach
        </ul>
    </div>
@endsection
```

`$article->category` လို့ပြောလိုက်ရင် လက်ရှိ Article Model နဲ့ သက်ဆိုင်တဲ့ Category တစ်ခုကို ရပါတယ်။ ရလာတဲ့ Category ရဲ့ `name` ကို ရိုက်ထုတ်ဖော်ပြထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

`$article-> comments` ဆိုရင်တော့ လက်ရှိ Article Model နဲ့ သက်ဆိုင်တဲ့ Comments စာရင်းကို ရပါတယ်။ အဲ့ဒီ Comments စာရင်းကို Loop လုပ်ပြီး ဖော်ပြထားတဲ့အတွက် ရလဒ်က အခုလို ဖြစ်မှာပါ။



လက်စနဲ့ `Comment` အသစ်တွေ ထပ်ထည့်လို့ ရအောင်နဲ့ ပြန်ဖျက်လို့ရအောင် လုပ်ပါမယ်။

`detail.blade.php` ကို အခုလို ထပ်မံဖြည့်စွက်ပါ။

```

@extends("layouts.app")

@section("content")
    <div class="container">
        <div class="card mb-2">
            ...
        </div>

        <ul class="list-group mb-2">
            <li class="list-group-item active">
                <b>Comments ({{ count($article->comments) }})</b>
            </li>
            @foreach($article->comments as $comment)
                <li class="list-group-item">
                    {{ $comment->content }}
                    <a href="{{ url('/comments/delete/' . $comment->id) }}"
                        class="close">
                        &times;
                    </a>
                </li>
            @endforeach
        </ul>

        <form action="{{ url('/comments/add') }}" method="post">
            <input type="hidden" name="article_id"
                value="{{ $article->id }}">
            <textarea name="content" class="form-control mb-2"
                placeholder="New Comment"></textarea>
            <input type="submit" value="Add Comment"
                class="btn btn-secondary">
        </form>
    </div>
@endsection

```

Comment တစ်ခုချင်းစီနဲ့အတူ `/comments/delete/{id}` Route ကိုသွားတဲ့ ခလုပ်တွေ ပါဝင်သွားပြီး၊ Comment စာရင်းရဲ့အောက်မှာ New Comment Form ပါဝင်သွားတာပါ။ Hidden Input တစ်ခုပါဝင်ပြီး သူရဲ့ Value က Article ID ဖြစ်တယ်ဆိုတာကို သတိပြုပါ။ Form ရဲ့ Action အရ ဒီ Form ကို Submit လုပ်လိုက်ရင် `/comments/add` Route ကို ရောက်သွားမှာပါ။ ဒါကြောင့် အဲ့ဒီ Route တွေ သွားထည့်ပေးရပါမယ်။

/routes/web.php မှာ အခုလို ဖြည့်ပေးပါ။

```
Route::post('/comments/add', 'CommentController@create');
Route::get('/comments/delete/{id}', 'CommentController@delete');
```

/comments/add အတွက် Method က post() ဆိုတာကို သတိပြုပါ။

မှတ်ချက် - Comment အသစ်အတွက် /articles/detail/{article_id}/comment/add ဆိုတဲ့ Route ကို သုံးရင် ပိုပြည့်စုံနိုင်ပါတယ်။ ဒါပေမယ့် Comment အတွက် Route ကို /comments နဲ့ သာ စလိုတဲ့အတွက် အဲ့ဒီနည်းကို မသုံးခဲ့တာပါ။ Route နဲ့ Controller ဆက်စပ်မှု Consistence ဖြစ်စေချင် လို့ပါ။

လက်ရှိနမူနာ Route နှစ်ခုလုံးက CommentController ကို ညွှန်းထားလို့ Comment Controller ဖိုင် ဆောက်ရပါမယ်။ ပြီးတဲ့အခါ create() နဲ့ delete() Method (၂) ခု ရေးပေးရပါမယ်။

```
php artisan make:controller CommentController
```

```
<?php

namespace App\Http\Controllers;

use App\Comment;
use Illuminate\Http\Request;

class CommentController extends Controller
{
    public function create()
    {
        $comment = new Comment;
        $comment->content = request()->content;
        $comment->article_id = request()->article_id;
        $comment->save();

        return back();
    }
}
```

```
public function delete($id)
{
    $comment = Comment::find($id);
    $comment->delete();

    return back();
}
```

App\Comment ကို Import လုပ်ထားတာ သတိပြုပါ။ create() နဲ့ delete() တို့ရဲ့ အလုပ်လုပ်ပုံ ကိုတော့ အထူးထပ်ပြီး ရှင်းပြဖို့ မလိုတော့ဘူးလို့ ထင်ပါတယ်။ ရေးထားတဲ့ကုဒ်မှာ အဓိပ္ပါယ်က ပေါ်လွင် နေပါပြီ။

အခုဆိုရင် ကျွန်တော်တို့ နမူနာအနေနဲ့ တည်ဆောက်နေတဲ့ Blog စနစ်လေးဟာ တော်တော်လေး အသက်ဝင်နေပါပြီ။ Article တွေ ထည့်လို့ရတယ်၊ ဖျက်လို့ရတယ်။ Comment တွေ တွဲပြပေးတယ်၊ Comment တွေ ထည့်လို့ရတယ်၊ ဖျက်လို့ရတယ်၊ စသဖြင့် အတော်လေး အဆင်ပြေနေပါပြီ။

နောက်တစ်ခန်းမှာ Authorization နဲ့ ပက်သက်တဲ့အကြောင်းအရာတွေ ထပ်ဖြည့်ကြပါမယ်။

အခန်း (၁၃) - Authorization

ဆော့ဖ်ဝဲလုံခြုံရေးနဲ့ ပက်သက်ရင် Authentication နဲ့ Authorization ဆိုတဲ့ အမည် ခပ်ဆင်ဆင်နှစ်ခု ရှိနေပါတယ်။ Authentication ဆိုတာ ဝင်ခွင့်ရှိမရှိ စစ်ဆေးခြင်းဖြစ်ပြီး၊ Authorization ဆိုတာကတော့ လုပ်ခွင့်ရှိမရှိ စစ်ဆေးခြင်းဖြစ်ပါတယ်။ အမည်နဲ့ သဘောသဘာဝ ဆင်ပေမယ့် တူတော့ မတူပါဘူး။ Laravel မှာ Authentication နဲ့ ပက်သက်လို့ လိုအပ်တဲ့ ကုဒ်တွေက ကိုယ်တိုင်ရေးစရာမလိုဘဲ Framework က ကြိုရေးပေးထားလို့ ထည့်ပုံထည့်နည်းကို ကြည့်ခဲ့ပြီးပါပြီ။ ဒါကြောင့် Login, Register, Logout စတဲ့လုပ်ငန်းတွေက ရရှိထားပြီး ဖြစ်ပါတယ်။

အခုဆက်လက်ပြီး Authorization နဲ့ပက်သက်တဲ့ နမူနာတွေ ရေးစမ်းကြည့်ပါမယ်။ ပထမအဆင့် အနေနဲ့ Article တွေထည့်တာ၊ ဖျက်တာ၊ Comment တွေ ထည့်တာ ဖျက်တာကို လူတိုင်းကို လုပ်ခွင့်မပေးဘဲ၊ Login ဝင်ထားတဲ့ သူကိုသာ လုပ်ခွင့် ပေးပါတော့မယ်။ ဒီအတွက် Auth Middleware ကို သုံးပြီး အလွယ် တစ်ကူ ရေးလို့ရပါတယ်။

Middleware အကြောင်းကို ကြားဖြတ်ပြီး နည်းနည်းပြောရရင်၊ Middleware ဆိုတာ Request Filter လို့ အလွယ်မှတ်နိုင်ပါတယ်။ Request တစ်ခုကို လက်ခံရရှိရင် ကြားဖြတ်စစ်ဆေးစီမံတဲ့အလုပ်တွေ လုပ်ပေး နိုင်တဲ့ နည်းပညာပါ။ Laravel မှာ CSRF Token, Auth စသဖြင့် Middleware တွေ အသင့် ပါပါတယ်။ CSRF Token Middleware က ကြားဖြတ်ပြီး စစ်နေလို့ Form တွေမှာ @csrf မပါရင် အလုပ်မလုပ်တာ ပါ။ ကိုယ်တိုင်လည်း Middleware တွေ ရေးလို့ရပါတယ်။ Third-party Middleware တွေလည်း လိုအပ် ရင် ထပ်ထည့်လို့ရပါတယ်။ ဒီစာအုပ်မှာတော့ ထူးခြားတဲ့ လိုအပ်ချက်မရှိလို့ ကိုယ်တိုင် Middleware တွေ ရေးသားပုံကို ထည့်မဖော်ပြပါဘူး။ လိုအပ်ရင် နောက်မှကိုယ့်ဘာသာ ဆက်လေ့လာရမှာပါ။

Auth Middleware ကို အသုံးပြုပြီး Login ဝင်ထားမထား စစ်ဖို့အတွက် Route မှာ စစ်လို့ရသလို Controller မှာလည်း စစ်လို့ရပါတယ်။ ဥပမာ -

```
Route::get('/articles/add', 'ArticleController@add')->middleware('auth');
```

ဒီလိုရေးပေးလိုက်ရင် /articles/add Route က Login ဖြစ်နေမှပဲ အလုပ်လုပ်တော့မှာပါ။ Login ဝင်မထားဘဲ သွားဖို့ကြိုးစားရင် Login Page ကို အလိုအလျှောက် ရောက်သွားမှာပါ။ ဒါပေမယ့် Route တစ်ခုချင်းစီမှာ အဲ့ဒီလိုလိုက်ရေးနေရရင် အလုပ်ရှုပ်ပါတယ်။ ဒါကြောင့် ArticleController Class ထဲမှာ အခုလိုဖြည့်စွက် ပေးသင့်ပါတယ်။

```
public function __construct()
{
    $this->middleware('auth')->except(['index', 'detail']);
}
```

Constructor ထည့်သွင်းလိုက်ခြင်းဖြစ်ပြီး Middleware အနေနဲ့ auth ကို အသုံးပြုဖို့ သတ်မှတ်ထားပါတယ်။ ဒါကြောင့် ဒီ Controller ထဲက လုပ်ဆောင်ချက်တွေကို Login ဖြစ်နေမှပဲ ပေးလုပ်တော့မှာပါ။ ဒါပေမယ့် except() နဲ့ index, detail နှစ်ခုကို ချန်ထားတဲ့အတွက် index() နဲ့ detail() တို့ကိုတော့ Login မဖြစ်လည်းဘဲ ခြွင်းချက်အနေနဲ့ အသုံးပြုခွင့်ပေးပါလိမ့်မယ်။ CommentController မှာတော့ အခုလို ရေးပေးသင့်ပါတယ်။

```
public function __construct()
{
    $this->middleware('auth');
}
```

သူ့မှာတော့ except() တွေဘာတွေ မလိုတော့ပါဘူး။ အခုဆိုရင် Login ဝင်မထားရင် ကြည့်ယုံပဲ ကြည့်လို့ရပြီး Login ဝင်ထားတော့မှသာ အသစ်ထည့်တာ၊ ဖျက်တာတွေ လုပ်လို့ရတော့မှာပါ။

တစ်ချို့ **+ Add Article** လိုခလုပ်တွေ အပါအဝင် တစ်ချို့ UI တွေကိုလည်း Login ဝင်ထားမှ ပြစေချင်ရင် ရပါတယ်။ `/resources/views/layouts` ဖိုင်ထဲက `app.blade.php` မှာ **+ Add Article** ခလုပ် ထည့်ထားတာ မှတ်မိဦးမှာပါ။ ဒီလိုပြင်ပေးလိုက်မယ်ဆိုရင် အဲဒီ **+ Add Article** ခလုပ်ကို Login ဝင်ထားမှ ပဲ မြင်ရတော့မှာပါ။

```
<li class="nav-item">
    @auth
        <a class="nav-link text-success"
            href="{{ url('/articles/add') }}">+ Add Article</a>
    @endauth
</li>
```

ပြောင်းပြန်အားဖြင့် Login မဝင်ထားမှ ပြစေချင်တာတွေ ရှိရင်လည်းရပါတယ်။ `@auth` အစား `@guest` ကို သုံးပေးရပါတယ်။

Authorizing Comment Delete (only owner)

ဒီတစ်ခါတော့ ကိုယ့်ထည့်ထားတဲ့ Comment ကိုပဲ ဖျက်ခွင့်ပြုတဲ့ ကုဒ်တွေ ထပ်ရေးပါမယ်။ စမ်းသပ် ရေးသားနိုင်ဖို့အတွက် Database Migration နဲ့ Seed ကို အရင်ပြင်ရပါမယ်။ ပထမဆုံးအနေနဲ့ `xxx_create_comments_table.php` ကိုဖွင့်ပြီး အခုလိုဖြည့်စွက်ပေးပါ။

```
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->id();
        $table->text('content');
        $table->integer('article_id');
        $table->integer('user_id');
        $table->timestamps();
    });
}
```

`user_id` Column ပါဝင်သွားတာပါ။ Comment တွေသိမ်းတဲ့အခါ သိမ်းတဲ့သူရဲ့ ID ကို တွဲသိမ်းနိုင်ဖို့ပါ။ ပြီးတဲ့အခါ `CommentFactory.php` မှာလည်း အခုလိုဖြည့်စွက်ပေးပါ။


```
$factory->define(Comment::class, function (Faker $faker) {
    return [
        "content" => $faker->paragraph,
        "article_id" => rand(1, 20),
        "user_id" => rand(1, 2),
    ];
});
```

user_id Column အတွက် တန်ဖိုးတစ်ခါတည်း ထည့်သတ်မှတ်ပေးလိုက်တာပါ။ ပြီးရင်တော့ DatabaseSeeder.php မှာ အခုလိုဖြည့်စွက်ပေးပါ။

```
public function run()
{
    factory(App\Article::class, 20)->create();
    factory(App\Category::class, 5)->create();
    factory(App\Comment::class, 40)->create();

    factory(App\User::class)->create([
        "name" => "Alice",
        "email" => "alice@gmail.com",
    ]);

    factory(App\User::class)->create([
        "name" => "Bob",
        "email" => "bob@gmail.com",
    ]);
}
```

Alice နဲ့ Bob ဆိုတဲ့ User (၂) ယောက်ကို တစ်ခါတည်း ထည့်လိုက်တာပါ။ UserFactory က Framework နဲ့အတူ နဂိုကတည်းက ပါဝင်ပြီးဖြစ်ပါတယ်။ ကိုယ့်ဘာသာ ထပ်ရေးပေးစရာ မလိုပါဘူး။ UserFactory ထဲမှာ name, email ကို Faker နဲ့ Random ပေးထားလို့ မသုံးချင်ပါဘူး။ ဒါပေမယ့် UserFactory ကို သွားပြင်စရာ မလိုပါဘူး။ နမူနာမှာ ပြထားတဲ့အတိုင်း DatabaseSeeder မှာ ကိုယ်ပေးချင်တဲ့ Property တွေကို တွဲပေးလို့ ရပါတယ်။ UserFactory ထဲမှာ Password အတွက် password ဆိုတဲ့စာလုံးကိုပဲ Hash ပြောင်းပြီး သိမ်းထားပေးပါတယ်။ ဒါကြောင့် User အားလုံးရဲ့ Password က password ဖြစ်ပါတယ်။

ပြီးတဲ့အခါ Migration နဲ့ Seed ကို အခုလို အတွဲလိုက် Run ပေးလိုက်ပါ။

```
php artisan migrate:fresh --seed
```

နောက်ကနေ `--seed` တွဲပေးထားတဲ့အတွက် `db:seed` ကို နောက်တစ်ကြောင်း ထပ် Run စရာမလိုတော့ပါဘူး။ ပြီးတဲ့အခါ `alice@gmail.com`, `bob@gmail.com` စတဲ့ အီးမေးလ် (၂) ခုထဲက နှစ်သက်ရာတစ်ခုနဲ့ Login ဝင်ပြီး စမ်းကြည့်လို့ရပါပြီ။ Password ကတော့ `password` ပါ။

လုပ်ချင်တာကတော့ Comment တွေကို လူတိုင်းကို ဖျက်ခွင့်မပေးဘဲ၊ မူလတင်ထားသူကိုသာ ဖျက်ခွင့်ပေးချင်တာပါ။ အဲ့ဒါမရေးခင် ဘယ် Comment ကို ဘယ်သူတင်ထားလဲ သိရဖို့ လိုပါသေးတယ်။ ဒါကြောင့် `/app` ဖို့ဒါအောက်က `Comment.php` မှာ အခုလို Relationship Method တစ်ခု ရေးပေးပါ။

```
public function user()
{
    return $this->belongsTo("App\User");
}
```

ပြီးရင် `/resources/views/articles` ဖို့ဒါအောက်က `detail.blade.php` မှာ အခုလိုပြင်ပေးပါ။

```

@extends("layouts.app")

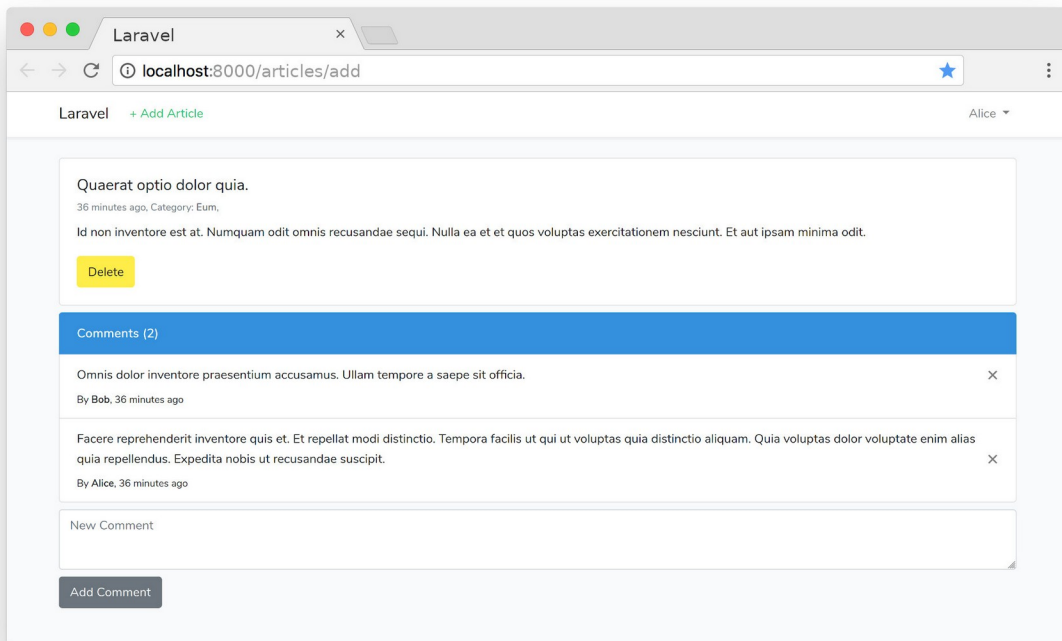
@section("content")
    <div class="container">
        ...

        <ul class="list-group mb-2">
            <li class="list-group-item active">
                <b>Comments ({{ count($article->comments) }})</b>
            </li>
            @foreach($article->comments as $comment)
                <li class="list-group-item">
                    {{ $comment->content }}
                    <a href="{{ url('/comments/delete/' . $comment->id) }}"
                        class="close">
                        &times;
                    </a>
                    <div class="small mt-2">
                        By <b>{{ $comment->user->name }}</b>,
                        {{ $comment->created_at->diffForHumans() }}
                    </div>
                </li>
            @endforeach
        </ul>

        @auth
            <form action="{{ url('/comments/add') }}" method="post">
                ...
            </form>
        @endauth
    </div>
@endsection

```

Comment ကိုဖော်ပြတဲ့အခါ Username နဲ့ Date Time ကိုပါ တစ်ခါတည်း ထည့်ပြလိုက်တာပါ။ ပြီးတော့ Comment Form ကိုလည်း @auth ဖြစ်နေမှပဲ ပြခိုင်းထားပါတယ်။ လက်ရှိရလဒ်က အခုလိုဖြစ်မှာပါ။



အခုဆိုရင် Comment တစ်ခုချင်းစီရဲ့အောက်မှာ ဘယ်သူရေးထားတာလဲဆိုတဲ့ နာမည်လေးတွေ ပေါ်နေပါပြီ။ ဆက်လက်ပြီး ကိုယ်ရေးထားတဲ့ Comment ကိုပဲ ဖျက်လို့ရအောင်၊ သူများ Comment တွေ ဖျက်လို့မရအောင် လုပ်ပါမယ်။

Laravel မှာ Authorization နဲ့ပတ်သက်ရင် Gate နဲ့ Policies လို့ခေါ်တဲ့ နည်းလမ်းနှစ်မျိုးပါပါတယ်။ သဘောသဘာဝကတော့ အတူတူပါပဲ။ Authorization Rule တွေနည်းရင် Gate နဲ့ပဲ ရေးလို့ရပါတယ်။ များရင်တော့ စုစည်းစည်း ဖြစ်သွားအောင် Policies အနေနဲ့ ရေးသင့်ပါတယ်။ အခုလက်ရှိနမူနာမှာတော့ တစ်ခုတည်းပဲ ရှိမှာမို့လို့ Gate နဲ့ပဲ ရေးမှာဖြစ်ပါတယ်။

တစ်ကယ်တော့ Gate တွေ Policies တွေမပါသေးဘဲ ဒီအတိုင်းလည်း စစ်လို့ရပါတယ်။ ဥပမာ - `CommentController` ရဲ့ `delete()` Method ကို အခုလို ပြင်နိုင်ပါတယ်။

```

public function delete($id)
{
    $comment = Comment::find($id);

    if($comment->user_id == auth()->user()->id) {
        $comment->delete();
        return back();
    } else {
        return back()->with('error', 'Unauthorized');
    }
}

```

Controller Method ထဲမှာပဲ Comment ရဲ့ user_id နဲ့ လက်ရှိ Login ဝင်ထားတဲ့ User ရဲ့ id တူမတူ စစ်လိုက်တာပါ။ auth() Function ကိုသုံးပြီး လက်ရှိ Login အခြေအနေကို ရယူနိုင်ပါတယ်။ ဥပမာ - auth()->check() က true ပြန်လာရင် Login ဖြစ်ပြီး false ပြန်လာရင် Login မဖြစ်ဘူးဆိုတဲ့ အဓိပ္ပါယ်ပါ။ လက်ရှိ Login ဝင်ထားတဲ့ User ကို လိုချင်ရင်တော့ auth()->user() နဲ့ ယူနိုင်ပါတယ်။ ဒီနည်းနဲ့ပဲ နမူနာမှာ Login User ရဲ့ id ကို ယူပြီးတိုက်စစ်ထားတာပါ။

ဒါပေမယ့် Controller Method တွေထဲမှာ အခုလို Authorization စစ်တဲ့ Logic တွေကို ဖြန့်ကျဲပြီး ရေးထားတာ အလေ့အကျင့်ကောင်း မဟုတ်ပါဘူး။ ကြာလာရင် စီမံရခက်လာပါလိမ့်မယ်။ ဒါကြောင့် Gate တို့ Policies တို့ကို သုံးရတာပါ။

Comment Delete အတွက် Authorization Logic ရေးဖို့အတွက် /app/Providers/ ဖိုဒါထဲက AuthServiceProvider.php ဖိုင်ကိုဖွင့်ပါ။ boot() Method ထဲမှာ အခုလိုရေးပေးပါ။

```

public function boot()
{
    $this->registerPolicies();

    Gate::define('comment-delete', function($user, $comment) {
        return $user->id == $comment->user_id;
    });
}

```

Gate Class ရဲ့ `define()` Method ကို အသုံးပြုပြီး Authorization Logic တစ်ခု သတ်မှတ်လိုက်တာပါ။ အမည်ကို `comment-delete` လို့ပေးထားပါတယ်။ ကြိုက်တဲ့အမည်ပေးလို့ရပါတယ်။ အဓိပ္ပါယ် ပေါ်လွင်အောင် ပေးထားတဲ့ သဘောပါ။ Logic ကိုတော့ နောက်က Function ထဲမှာ ဆက်ရေးထားပါတယ်။ Function က Parameter နှစ်ခုလက်ခံပါတယ်။ `$user` နဲ့ `$comment` ပါ။ Logic ကတော့ ရိုးရိုး လေးပါ `$user->id` နဲ့ `$comment->user_id` တူရင် မှန်တယ်လို့ သတ်မှတ်လိုက်တာပါ။

ဒီ Gate ကို အသုံးပြုပြီး `CommentController` ရဲ့ `delete()` Method ကို ပြင်ရေးပါမယ်။ မရေးခင် အပေါ်မှာ Gate Class ကို အခုလိုအရင် Import လုပ်ပေးပါ။

```
use Illuminate\Support\Facades\Gate;
```

`delete()` Method အတွက် ကုဒ်က ဒီလိုပါ။

```
public function delete($id)
{
    $comment = Comment::find($id);

    if( Gate::allows('comment-delete', $comment) ) {
        $comment->delete();
        return back();
    } else {
        return back()->with('error', 'Unauthorize');
    }
}
```

စောစောက ကုဒ်နဲ့အတူတူပါပဲ။ ကိုယ့်ဘာသာ စစ်မယ့်အစား Gate ရဲ့ အကူအညီနဲ့ ရေးပြီးသား `comment-delete` ကို လှမ်းခေါ်လိုက်တဲ့ သဘောမျိုးပါ။ ဒီလိုရေးရင်လည်း ရပါတယ်။

```
public function delete($id)
{
    $comment = Comment::find($id);

    if(Gate::denies('comment-delete', $comment)) {
        return back()->with('error', 'Unauthorized');
    }

    $comment->delete();
    return back();
}
```

Gate ရဲ့ `allows()` သို့မဟုတ် `denies()` Method ကို အသုံးပြုပြီး လက်ရှိအလုပ်ကို လုပ်ခွင့်ရှိမရှိ စစ်လို့ရတဲ့သဘော ဖြစ်ပါတယ်။ Gate ကို `define()` နဲ့ သတ်မှတ်ခဲ့စဉ်က Function Parameter မှာ `$user` ကိုလည်း ထည့်ပေးခဲ့ရပါတယ်။ ဒါပေမယ့် ပြန်သုံးတဲ့အချိန်မှာ User ကို ထည့်ပေးစရာမလိုပါဘူး။ Laravel က သူ့ဘာသာ ထည့်ပေးသွားပါတယ်။ ဒါကြောင့် ပြန်သုံးတဲ့အချိန်မှာ `$comment` တစ်ခုတည်းကိုပဲ ထည့်ပေးထားတာပါ။

ဒီနည်းနဲ့ Laravel မှာ ဘယ်သူဘယ်အလုပ်လုပ်ခွင့်ရှိတယ်ဆိုတဲ့ Authorization နဲ့ Access Control ကို စီမံရေးသားရတာဖြစ်ပါတယ်။ မေ့ကျန်ခဲ့မှာစိုးလို့ `CommentController` ရဲ့ `create()` Method ကိုလည်း အခုလို ပြင်ပေးဖို့လိုအပ်ပါတယ်။

```
public function create()
{
    $comment = new Comment;
    $comment->content = request()->content;
    $comment->article_id = request()->article_id;
    $comment->user_id = auth()->user()->id;
    $comment->save();

    return back();
}
```

ဒီတော့မှ `Comment` တွေသိမ်းတဲ့အခါ `user_id` ကို ထည့်သိမ်းသွားမှာပါ။ `Article` တွေကိုလည်း အဲ့လိုပဲ လူတိုင်းဖျက်လို့မရဘဲ၊ ရေးထားတဲ့သူပဲ ဖျက်လို့ရအောင် ကိုယ့်ဘာသာ စမ်းလုပ်ကြည့်သင့်ပါတယ်။

အခန်း (၁၄) – Basic API

API (Application Program Interface) ဆိုတာဟာ UI မပါတဲ့ကုဒ်လို့ ဆိုနိုင်ပါတယ်။ API ကုဒ်နဲ့ တစ်ခြား Application ကုဒ်ဟာ အခြေခံတူညီပြီး UI ပါခြင်းနဲ့ မပါခြင်းသာ ကွာသွားတာပါ။ Application က Input Data ကို လက်ခံပြီး UI ကို Output အနေနဲ့ ပြန်ပြပေးပါမယ်။ API ကတော့ Input Data ကို လက်ခံပြီး Output ကိုလည်း Data အနေနဲ့ပဲ ပြန်ပေးပါတယ်။ UI မပါတဲ့အတွက် ရလာတဲ့ အားသာချက်တွေကတော့

- ၁။ UI ကို နှစ်သက်ရာ နည်းပညာနဲ့ ခွဲခြားရေးသားနိုင်ခြင်း ဖြစ်ပါတယ်။ ဥပမာ – PHP API ကို JavaScript UI နဲ့ တွဲသုံးလို့ ရတဲ့သဘောပါ။
- ၂။ Platform အမျိုးမျိုးအတွက် UI အမျိုးမျိုးခွဲရေးထားနိုင်ပါတယ်။ ဥပမာ – Web App, Android App, iOS App စသဖြင့် အမျိုးမျိုးက API တစ်ခုတည်းကို ဆက်သွယ်အသုံးပြု အလုပ်လုပ်နိုင်ခြင်း ဖြစ်ပါတယ်။

တစ်ကယ်တော့ မျက်စိထဲမှာ မြင်အောင် UI လို့ ပြောနေတာပါ။ ဒီ UI ပရိုဂရမ်တွေကို Frontend လို့လည်း ခေါ်ကြပါတယ်။ Client လို့လည်း ခေါ်ကြပါတယ်။ API ကိုတော့ Backend လို့ ခေါ်ကြပါတယ်။

ဒါကြောင့် Laravel ကို အသုံးပြုပြီး API ဖန်တီးတဲ့အခါ တစ်ခြား သဘောသဘာဝတွေ အများကြီး ပြောင်းလဲခြင်းမရှိဘဲ၊ View Template တွေကို မသုံးတော့တာပဲ ရှိတယ်လို့ အလွယ်ပြောနိုင်ပါတယ်။ အခြေခံ Web နည်းပညာတွေထဲမှာ တစ်ခုအပါအဝင်ဖြစ်တဲ့ Session ကိုလည်း API တွေမှာ သုံးလေ့မရှိပါဘူး။ ဒါကြောင့် Authentication နဲ့ Authorization ပိုင်းမှာတော့ ရိုးရိုး Application နဲ့ API တော်တော်

ကွာသွားပါလိမ့်မယ်။ လောလောဆယ် အဲဒီလောက်ထိ ခေါင်းစားမခံပါနဲ့ဦး။ အခြေခံလေးတွေပဲ အရင် ကြေညက်အောင် ကြည့်လိုက်ကြပါစို့။

API Route

Routing အကြောင်း ရှင်းပြခဲ့တုန်းက လိုက်နာသင့်တဲ့ URL Pattern တွေကိုပြောခဲ့တာ မှတ်မိဦးမှာပါ။

- `/resource/action/id`
- `/resource/action/id/sub-resource/sub-action`

API အကြောင်းပြောတဲ့အခါ အရင်ဆုံးဒီကနေစပြောရပါလိမ့်မယ်။ URL Pattern မပြောင်းပါဘူး။ ဒါပေမယ့် action မလိုတော့ပါဘူး။ ဒါကြောင့် API အတွက် သုံးမယ့် URL Pattern က ဒီလိုပါ။

- `/resource/id`
- `/resource/id/sub-resource`

Action အစား HTTP Method တွေဖြစ်ကြတဲ့ GET, POST, PUT, PATCH, DELETE တို့ကို အသုံးပြုသွားရမှာပါ။ ဒီနည်းက REST (Representational State Transfer) လို့ခေါ်တဲ့ နည်းစနစ်ကနေလာပြီး API ဖန်တီးသူတိုင်း စံထားပြီး သုံးနေကြတဲ့ နည်းပါ။ အရင်တုန်းကတော့ REST ရဲ့ အားသာချက်တွေ ဘယ်လိုဘယ်ဝါရှိတယ်၊ ဒါကြောင့် သုံးသင့်တယ် စသဖြင့် ရှင်းပြရမှာပါ။ အခုတော့ အဲဒီလောက် ပြောနေစရာ မလိုတော့ပါဘူး။ REST ဆိုတာ မဖြစ်မနေ သုံးကိုသုံးရမယ့် နည်းစနစ်ဖြစ်နေပါပြီ။ နောက်ကွယ်မှာ ကျယ်ပြန့်တဲ့ သဘောသဘာဝတွေ ရှိပေမယ့် လက်တွေ့အသုံးပြုနိုင်ဖို့ဒီ URL Pattern ကို မှတ်ထားရင် လုံလောက်နေပါပြီ။

API Route တွေကို Manual သတ်မှတ်မယ်ဆိုရင် ဒီလိုပုံစံဖြစ်နိုင်ပါတယ်။

```
Route::get('/categories', 'CategoryApiController@index');
Route::get('/categories/{id}', 'CategoryApiController@detail');
Route::post('/categories', 'CategoryApiController@create');
Route::put('/categories/{id}', 'CategoryApiController@update');
Route::patch('/categories/{id}', 'CategoryApiController@update');
Route::delete('/categories/{id}', 'CategoryApiController@delete');
```

URL က နှစ်ခုတည်းပါ။ `/categories` နဲ့ `/categories/{id}` ဖြစ်ပါတယ်။ ဒါပေမယ့် `get`, `post`, `put`, `patch`, `delete` တို့နဲ့တွဲလိုက်တဲ့အခါ အလုပ် (၅) ခုရသွားပါတယ်။ (၆) ကြောင်း ရေးထားပေမယ့် `put` နဲ့ `patch` အတွက် `Controller Method` တစ်ခုတည်းကို ညွှန်းထားတာကို သတိပြုပါ။ အခြေခံအားဖြင့် `get()` ကို `Data` တွေ ရယူတဲ့ လုပ်ငန်းအတွက် သုံးပါတယ်။ `post()` ကို `Data` အသစ်တည်ဆောက်တဲ့ လုပ်ငန်းအတွက်သုံးပါတယ်။ `put()` နဲ့ `patch()` ကို `Data` ပြင်ဆင်တဲ့ လုပ်ငန်းအတွက် သုံးပါတယ်။ တူတယ်လို့ ပြောလို့ရသလို၊ မတူဘူးလို့လည်း ပြောလို့ရပါတယ်။ `PUT` ဆိုတာ နဂို `Data` ကို `Data` အသစ်နဲ့ အစားထိုး ပြင်ဆင်တဲ့ ပြင်ဆင်မှုမျိုးမှာ သုံးရတာပါ။ `PATCH` ကိုတော့ နဂို `Data` ထဲက တစ်စိတ်တစ်ပိုင်းကို ရွေးထုတ်ပြင်ဆင်တဲ့ လုပ်ငန်းမျိုးမှာ သုံးရတာပါ။ ပြင်တာချင်းတူပေမယ့် သဘောသဘာဝ ကွာပါတယ်။ ဒါပေမယ့် တစ်ချို့တွေလည်း အဲ့ဒီလောက်ထိ အသေးစိတ် ခွဲမနေပါဘူး။ ပြင်တဲ့အလုပ်ဆိုရင် ဘယ်လိုပဲပြင်ပြင် `PUT` သို့မဟုတ် `PATCH` နှစ်ခုထဲက နှစ်သက်ရာ သုံးလိုက်ကြတာပါပဲ။ `delete()` ကိုတော့ `Data` တွေပယ်ဖျက်တဲ့ လုပ်ငန်းမှာ အသုံးပြုပါတယ်။

`API Route` တွေရဲ့ သဘောသဘာဝကို သိအောင်သာပြောတာပါ။ `Laravel` မှာ အဲ့ဒီလိုတစ်ကြောင်းချင်း ကိုယ်တိုင်သတ်မှတ်ပေးစရာမလိုပါဘူး။ ဒီလိုရေးလိုက်ရင် ရပါတယ်။

```
Route::apiResource('/categories', 'CategoryApiController');
```

ဒါဆိုရင် `Laravel` က လိုအပ်တဲ့ `get`, `post`, `put`, `delete` `ROUTE` တွေ အကုန်လုံးကို အလိုအလျှောက် သတ်မှတ်ပေးသွားမှာပါ။ ဒါကြောင့် `/routes` ဖိုင်ထဲက `api.php` ကိုဖွင့်ပြီး အပေါ်မှာပေးထားတဲ့ ကုဒ်ကို ရေးဖြည့်ပေးလိုက်ပါ။ **သတိပြုပါ** - အခုသုံးတာ `web.php` မဟုတ်တော့ပါဘူး။ `api.php` ဖြစ်သွားပါပြီ။

`web.php` နဲ့ `api.php` ဘာကွာလဲဆိုတော့၊ `web.php` ထဲမှာ ရေးထားတဲ့ `Route` တွေကို လိုအပ်ရင် `CSRF` စစ်ပြီး `Session` အသုံးပြုခွင့် ပေးထားပါတယ်။ `api.php` ထဲက `Route` တွေကိုတော့ `CSRF` မစစ်တော့ပါဘူး။ `Session` လည်းသုံးခွင့်မပေးတော့ပါဘူး။ ဖြည့်စွက်ချက်အနေနဲ့ `Rate Limit` ပါဝင်သွားပါတယ်။ ဒီ `API` ကို ခေါ်သုံးတဲ့ `Client` တွေဟာ တစ်မိနစ်မှာ ဘယ်နှစ်ကြိမ်သာ ခေါ်ခွင့်ရှိတယ်ဆိုတဲ့ ကန့်သတ်ချက်ပါ။ ပြီးတော့ `api.php` ထဲမှာ ရေးထားတဲ့ `Route` တွေကို အသုံးပြုဖို့ ရှေ့က `/api` ထည့်ပြီး သုံးပေးရပါတယ်။ ဒါကြောင့် `Route` မှာ လိပ်စာကို `/categories` လို့ ပေးထားပေမယ့် အသုံးပြုတဲ့

အခါ `/api/categories` လို့ သုံးပေးရမှာပါ။

ဆက်လက်ပြီးတော့ `CategoryApiController` အမည်နဲ့ `Controller` တစ်ခုတည်ဆောက်ပါမယ်။ ဒီလို `Run` ပေးပါ။

```
php artisan make:controller CategoryApiController --api --model=Category
```

ထူးခြားချက်အနေနဲ့ `--api` ပါဝင်သွားသလို `--model=Category` လည်း ပါဝင်သွားပါတယ်။ `--api` လို့ ထည့်ပေးလိုက်တဲ့အတွက် `Controller` ဖိုင်ထဲမှာ `index`, `store`, `show`, `update`, `destroy` ဆိုတဲ့ `Method` (၅) ခု တစ်ခါတည်း ပါဝင်သွားမှာပါ။ `--model` နဲ့ `Category` ကို တွဲပေးထားလို့ အထဲမှာ `Category Model Class` ကို အသင့် `Import` လုပ်ထားပေးမှာပါ။ ကိုယ့်ဘာသာလုပ်လည်း ရပေမယ့် အခုလို ဖိုင်တည်ဆောက် ကတည်းက ထည့်ခိုင်းလိုက်လို့ ရတယ်ဆိုတာကို သိစေချင်လို့ပါ။

လက်ရှိရေးထားတဲ့ `Route` နဲ့ `Controller` အရ အလုပ်လုပ်တဲ့အခါ အခုလို အလုပ်လုပ်သွားမှာပါ။

- **GET** `/categories` -> **index()**
- **GET** `/categories/{id}` -> **show()**
- **POST** `/categories` -> **store()**
- **PUT** `/categories/{id}` -> **update()**
- **DELETE** `/categories/{id}` -> **destroy()**

ကျွန်တော်တို့ ကိုယ့်ဘာသာပေးခဲ့တဲ့ `Method` အမည်တွေနဲ့ `Laravel` ကပေးတဲ့ `Default Method` အမည်တွေ နည်းနည်း ကွာပါတယ်။ ဒါပေမယ့် မှတ်ရခက်လောက်အောင် ကွာတာမျိုးတော့ မဟုတ်လို့ အဆင်ပြေမယ်လို့ ယူဆပါတယ်။ လိုအပ်တဲ့ကုဒ်တွေ စရေးပါမယ်။

```
<?php

namespace App\Http\Controllers;

use App\Category;
use Illuminate\Http\Request;

class CategoryApiController extends Controller
{
    public function index()
    {
        return Category::all();
    }

    public function store()
    {
        $category = new Category;
        $category->name = request()->name;
        $category->save();

        return $category;
    }

    public function show($id)
    {
        return Category::find($id);
    }

    public function update($id)
    {
        $category = Category::find($id);
        $category->name = request()->name;
        $category->save();

        return $category;
    }

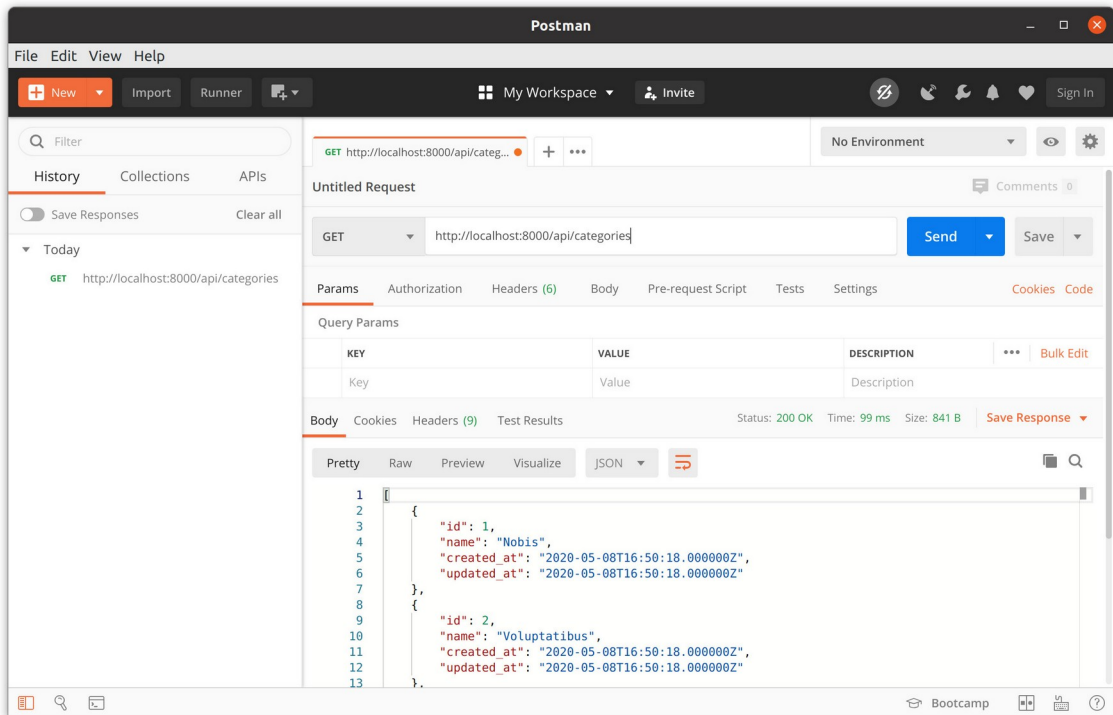
    public function destroy($id)
    {
        $category = Category::find($id);
        $category->delete();

        return $category;
    }
}
```

ကုဒ်က ရိုးရိုးရှင်းရှင်းပါပဲ။ `index()` က Category အားလုံးကို Model Collection အနေနဲ့ ပြန်ပေးပါတယ်။ Laravel က အဲ့ဒီ Model Collection ကို JSON Response ဖြစ်အောင် အလိုအလျှောက် ပြောင်းပြီး ပြန်ပေးပါတယ်။ ဒါကြောင့် Model Collection ကို JSON ဖြစ်အောင် Encode လုပ်တဲ့ အလုပ်တွေ၊ Response Status Code သတ်မှတ်တဲ့အလုပ်တွေ၊ Response Header မှာ Content-Type သတ်မှတ်တဲ့ အလုပ်တွေ၊ တစ်ခုမှ လုပ်စရာမလိုတော့ပါဘူး။ တစ်ကယ်တော့ API Request / Response ပိုင်းမှာ ကိုယ့်ဘာသာ လုပ်ရမယ်ဆိုရင် တော်တော်အလုပ်ရှုပ်တာပါ။ အခု အဲ့လောက်အလုပ်ရှုပ်တဲ့ ကိစ္စကို လွယ်လွယ်လေးနဲ့ ရနေတာပါ။

`view()` Method ကတော့ `id` နဲ့ကိုက်တဲ့ Category Model တစ်ခုကို ပြန်ပေးပါတယ်။ အတူတူပါပဲ။ Laravel က JSON Response အနေနဲ့ပြောင်းပေးလိုက်မှာပါ။ `store()` က Request Data name ကို အသုံးပြုပြီး Model အသစ်ဆောက်ပေးပါတယ်။ ရလာတဲ့ Model ကို ပြန်ပေးပါတယ်။ `update()` က Request Data name ကို အသုံးပြုပြီး `id` နဲ့ကိုက်တဲ့ Model ကို Update လုပ်ပေးပါတယ်။ `destroy()` ကတော့ `id` နဲ့ ကိုက်တဲ့ Model ကို ဖျက်ပေးပါတယ်။ ဒါဟာ Category တွေကို စီမံလိုရတဲ့ အခြေခံ API တစ်ခုကို အလွယ်တစ်ကူနဲ့ မြန်မြန်ဆန်ဆန် ရရှိသွားခြင်းပဲ ဖြစ်ပါတယ်။

ရေးထားတဲ့ API လုပ်ဆောင်ချက်တွေကို စမ်းဖို့အတွက် cURL, Postman, Insomnia စသဖြင့် အသုံးဝင်တဲ့ API Testing Tool အမျိုးမျိုးရှိပါတယ်။ နမူနာအနေနဲ့ အဲ့ဒီထဲက Postman ကိုအသုံးပြုဖော်ပြပါမယ်။ ဒါကြောင့် getpostman.com ကနေ Postman ကို Download လုပ်ပြီး Install လုပ်ထားဖို့လိုပါမယ်။

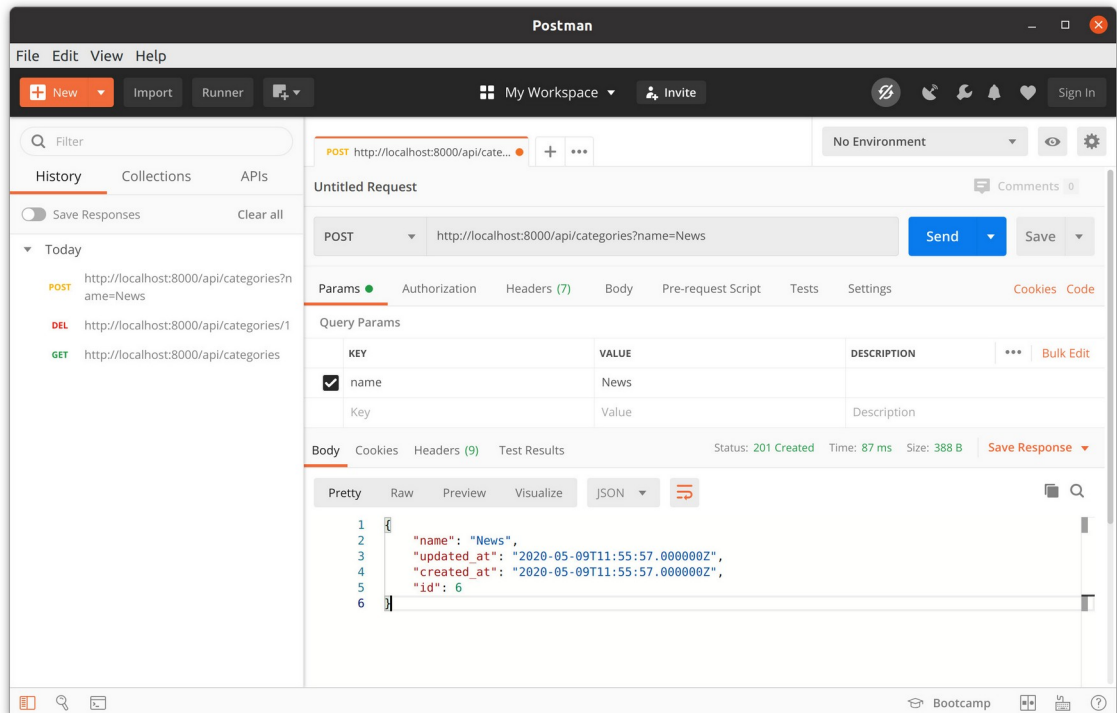


အပေါ်က နမူနာပုံကိုကြည့်ပါ။ Request Method ရွေးရတဲ့နေရာမှာ GET ကိုရွေးထားပြီး URL လိပ်စာ အနေနဲ့ `http://localhost:8000/api/categories` ကို ပေးထားပါတယ်။ ပြီးတဲ့အခါ Send နှိပ်လိုက်ရင် Postman က Request ကို ကျွန်တော်တို့ ရေးထားတဲ့ API ထံ ပေးပို့သွားပြီး ပြန်ရလာတဲ့ Response Data ကို ဖော်ပြပေးမှာ ဖြစ်ပါတယ်။

`php artisan serve` နဲ့ ပရောဂျက်ကို Run ထားဖို့တော့လိုပါတယ်။ Postman မသုံးဘဲ Browser မှာပဲ လိပ်စာအပြည့်အစုံ ရိုက်ထည့်ရင်လည်း ရပါတယ်။ ရိုးရိုး GET Request တွေ အတွက်က Browser နဲ့ တင် အဆင်ပြေပါတယ်။ တစ်ခြား POST, PUT, DELETE တွေသာ Browser မှာ စမ်းရခက်တာပါ။

Postman URL လိပ်စာနေရာမှာ `http://localhost:8000/api/categories/1` လို့ပြောင်းပြီး စမ်းကြည့်ရင်တော့ `show()` Method အလုပ်လုပ်သွားမှာဖြစ်လို့ id နံပါတ် 1 နဲ့ကိုက်ညီတဲ့ Category Data ကို ပြန်လည်ရရှိမှာ ဖြစ်ပါတယ်။ Request Method မှာ DELETE ကိုရွေးပြီး `http://localhost:8000/api/categories/1` ကို Request ပေးပို့ရင်တော့ id နံပါတ် 1 နဲ့ ကိုက်ညီတဲ့ Category ပျက်သွားမှာဖြစ်ပါတယ်။ စမ်းကြည့်လို့ရပါတယ်။

အသစ်ထည့်ပြီးစမ်းကြည့်ချင်ရင်တော့ Request Method မှာ POST ကို ရွေးထားပြီး `http://localhost:8000/api/categories` ကို Request ပေးပို့ရမှာပါ။ name Parameter ပါဖို့လိုပါတယ်။ မပါရင် Error တက်မှာပါ။ Validation စစ်တဲ့ကုဒ် မရေးထားပါဘူး။ အောက်ကနမူနာပုံမှာကြည့်ပါ။ Params အနေနဲ့ name ထည့်ပေးထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။



အသေးစိတ်ထပ်ပြောမယ်ဆိုရင်တော့ API Authentication / Authorization နဲ့ ပတ်သက်တဲ့ အကြောင်းတွေ၊ API Design နဲ့ ပတ်သက်တဲ့အကြောင်းတွေ ပြောစရာရှိပါသေးတယ်။ ဒီစာအုပ်မှာတော့ ထည့်မပြောနိုင်တော့ပါဘူး။ ပြောဖို့မှာလည်း နည်းနည်းတော့ စောပါသေးတယ်။ ပါဝင်တဲ့ အခြေခံတွေကိုသာ အရင်ကြေညက်အောင် ကြိုးစားထားသင့်ပါတယ်။

ဒီအဆင့်ထိ ရေးခဲ့သမျှကုဒ် အပြည့်အစုံကို လိုအပ်တယ်ဆိုရင် အောက်မှာပေးထားတဲ့ လိပ်စာကနေ Download ရယူနိုင်ပါတယ်။

- <https://github.com/eimg/laravel-book>

အခန်း (၁၅) - Deployment

Laravel ကိုအသုံးပြုပြီး ရေးပုံရေးနည်းတွေ ပြောပြီးပြီဆိုတော့၊ ရေးပြီးသားပရောဂျက်ကို အများအသုံးပြု နိုင်ဖို့ Publish တော့မယ်ဆိုရင် သတိပြုသင့်တဲ့ အချက်တွေကို ဖော်ပြသွားပါမယ်။

၁။ Laravel ပရောဂျက်တွေမှာ ကုဒ်ရေးတဲ့အခါ Route, Controller, View စသဖြင့် သူနေရာနဲ့သူ ရေး ရပေမယ့် နောက်ဆုံးရလဒ်ကတော့ `/public` ဖိုဒါ ဖြစ်ပါတယ်။ ဒါကြောင့် ပရောဂျက်ကို Publish လုပ်ဖို့ Web Server တစ်ခုနဲ့ Setup လုပ်တဲ့အခါ ပရောဂျက်ကြီးတစ်ခုလုံးကို Web Document Root အနေနဲ့ သတ်မှတ်ရမှာမဟုတ်ဘဲ `/public` ဖိုဒါကိုပဲ Root အနေနဲ့ သတ်မှတ်ပေးရမှာ ဖြစ်ပါတယ်။ ဒါဟာ ကောင်းမွန်တဲ့ ဖွဲ့စည်းပုံတစ်ခုပါ။ Web Server ကနေတစ်ဆင့် ရလဒ်ကိုသာ Access လုပ်လို့ရမှာ ဖြစ်ပြီး၊ `/public` ဖိုဒါ အပြင်ဘက်က Framework Source Code နဲ့ တစ်ခြား Route, Controller, View, Model ကုဒ်တွေကို Web Server ကနေတစ်ဆင့် Access လုပ်လို့ရမှာ မဟုတ်ပါဘူး။

၂။ `.env` ဖိုင်ကို သတိထားပါ။ `.env` ဖိုင် (၂) ခုရှိသင့်ပါတယ်။ Setting တွေ မတူတဲ့အတွက် ကိုယ့် စက်ထဲက `.env` ဖိုင်နဲ့ Server ပေါ်က `.env` ဖိုင် တူမှာမဟုတ်ပါဘူး။ ဥပမာ အားဖြင့် ကိုယ့်စက်ထဲက `.env` ဖိုင်မှာ `APP_ENV` က `local` ဖြစ်နေပေမယ့် Server ပေါ်က `.env` မှာ `production` ဖြစ်သင့် ပါတယ်။ ကိုယ့်စက်ထဲမှာ `APP_URL` က `localhost` ဖြစ်နေပေမယ့် Server ပေါ်မှာ Domain Name အမှန်ဖြစ်သင့်ပါတယ်။ `APP_DEBUG` က `true` ဆိုရင် တစ်ခုခုအဆင်မပြေ တဲ့အခါ Error အပြည့်အစုံပြ မှာပါ။ Server ပေါ်မှာ `false` ဖြစ်နေသင့်ပါတယ်။ ဒါမှ User ကို Error တွေအကုန် လျှောက်မပြတော့မှာ ပါ။ `DB_USERNAME` တို့ `DB_PASSWORD` တို့ဟာလည်း ကိုယ့်စက်ထဲက Setting နဲ့ Server ပေါ်က Setting တူမှာမဟုတ်ပါဘူး။ ဒါကြောင့် `.env` ဖိုင်ကို Server ရဲ့ Setting ပေါ်မူတည်ပြီး လိုအပ်သလို ပြင်ဆင်ပေးဖို့ လိုအပ်ပါတယ်။

၃။ Publish မလုပ်ခင် ပရောဂျက်ဖိုဒါထဲမှာ ဒီ Command ကို Run ပေးသင့်ပါတယ်။

```
composer install -o --no-dev
```

Namespaces အခန်းမှာ Namespace Import လုပ်လိုက်တာနဲ့ သက်ဆိုင်ရာဖိုင်ကို အလိုအလျောက် `include()` လုပ်ပေးအောင် လုပ်ထားလို့ရတယ်လို့ ပြောခဲ့ဖူးပါတယ်။ Composer မှာ အဲ့ဒီလို အလုပ် လုပ်ပေးနိုင်တဲ့ `autoload` လုပ်ဆောင်ချက် ပါဝင်ပြီး Laravel က အသုံးပြုထားပါတယ်။ ဒီအလုပ် လုပ် နိုင်ဖို့အတွက် Namespace Import လုပ်တိုင်း Composer က Class ဖိုင်ကို လိုက်ရှာရပါတယ်။ `composer install` ကို `-o` Option နဲ့ Run တဲ့အခါ ဖိုင်ကို လိုက်ရှာနေစရာ မလိုအောင် Cache လုပ် ထားလိုက်လို့ အလုပ်လုပ်ပုံ ပိုမြန်သွားမှာဖြစ်ပါတယ်။ `--no-dev` ရဲ့ အဓိပ္ပါယ်ကတော့ Development Dependency ခေါ် Test Library တွေ Build Library တွေကို ထည့်လုပ်စရာမလိုဘူးလို့ ပြောလိုက်တာပါ။

၄။ ပြီးတဲ့အခါ Framework ကုဒ်တွေ အတွက်လည်း Cache တွေထုတ်ပေးရပါမယ်။ ဒီလိုပါ -

```
php artisan config:cache
php artisan route:cache
php artisan view:cache
```

`/config` ဖိုဒါထဲက ကုဒ်တွေကို နမူနာတွေမှာ ထိစရာ ပြင်စရာ မလိုခဲ့ပေမယ့် ဖွင့်ကြည့်လို့ ရပါတယ်။ App Setting တွေ Database Setting တွေ Auth Setting တွေ အတွက် ဖိုင်တွေအများကြီး ရှိတယ်ဆိုတာ ကို တွေ့ရပါလိမ့်မယ်။ `config:cache` က အဲ့ဒီဖိုင်တွေ အားလုံးကို ပေါင်းပေးလိုက်တာပါ။ ဒါကြောင့် အလုပ်လုပ်တဲ့အခါ ဖိုင်တစ်ခုချင်းစီကို လိုက်ဖတ်စရာမလိုတော့ ပိုမြန်သွားပါလိမ့်မယ်။ အတူတူပါပဲ `route:cache` ကလည်း ရေးထားသမျှ Route တွေအကုန်လုံးကို Function တစ်ခုတည်းဖြစ်အောင် ပေါင်းပေးလိုက်တာပါ။ ဒါကြောင့် Route (၁၀၀) ရှိလို့ Route Method တွေကို အကြိမ် (၁၀၀) Run စရာမ လိုတော့ဘဲ တစ်ကြိမ်းတည်းနဲ့ အလုပ်လုပ်ပေးသွားမှာပါ။ `view:cache` ကတော့ Blade ရေးထုံးအတိုင်း ရေးထားတဲ့ View Template တွေကို ရိုးရိုး PHP ဖြစ်အောင် ပြောင်းပေးသွားမှာမို့လို့ တစ်ကြိမ် အလုပ် လုပ်တိုင်း တစ်ခါပြောင်းပြီး လုပ်နေစရာမလိုတော့ပါဘူး။

တစ်ခုတော့ သတိပြုပါ။ ဒီလို Cache တွေ ထုတ်ထားပြီးတော့မှ Config တွေ Route တွေ View တွေကို ပြင်ခဲ့ရင် အလုပ်လုပ်မှာ မဟုတ်တော့ပါဘူး။ Cache ကိုပဲ အသုံးပြုမှာမို့လို့ ကိုယ့်ပြင်ဆင်မှုက သက်ရောက်မှုရှိမှာ မဟုတ်တော့ပါဘူး။ ဒါကြောင့် လိုအပ်ရင် Cache တွေကို ပြန်ရှင်းလို့လည်း ရပါတယ်။ ဒီလိုပါ -

```
php artisan config:clear
php artisan route:clear
php artisan view:clear
```

ဒီလောက်ဆိုရင် ဒီစာအုပ်မှာ ဖော်ပြချင်တဲ့ အကြောင်းအရာ စုံသွားပြီဖြစ်ပါတယ်။

ဆက်လက်ပြီးလေ့လာမယ်ဆိုရင် Framework Architecture ပိုင်းက Service Container နဲ့ Service Provider လို လုပ်ဆောင်ချက်မျိုးတွေ၊ Database ပိုင်းက Query Builder လို လုပ်ဆောင်ချက်မျိုးတွေ၊ Frontend ပိုင်းအတွက် Laravel Mix လို ကိစ္စမျိုးတွေ၊ API အတွက် Laravel Passport လို ကိစ္စမျိုးတွေ၊ Logging နဲ့ Task Scheduling လို ကိစ္စမျိုးတွေ ကျန်ပါသေးတယ်။ သိပ်ပြီးတော့ ရှေ့လော မကြီးစေချင်ပါဘူး။ အခုဖော်ပြခဲ့တဲ့ အခြေခံတွေကိုသာ ကောင်းကောင်း ကြေညက်ပိုင်နိုင်အောင် စောက်ချပြီး အရင်လုပ်ပါ။ အခြေခံတွေ ပိုင်နိုင်ပြီဆိုတဲ့အခါ ဒီလို ပိုအဆင့်မြင့်တဲ့အပိုင်းတွေကို လေ့လာတဲ့အခါ အများကြီး အဆင်ပြေသွားပါလိမ့်မယ်။

Laravel ရဲ့ အားသာချက်ကတော့ လေ့လာစရာ Resource တွေပေါများခြင်းပဲဖြစ်ပါတယ်။ အဲ့ဒီထဲမှာ တစ်ခုအပါအဝင်က Laracasts ဖြစ်ပါတယ်။ Laracasts ဟာ Laravel အတွက်သာမက ဆက်စပ်လိုအပ်တဲ့ PHP, JavaScript နဲ့ ပက်သက်တဲ့ အကြောင်းအရာတွေကို အခြေခံကနေ အဆင့်မြင့်ထိ ဗွီဒီယိုသင်ခန်းစာ အစုံရှိတဲ့ Platform တစ်ခုဖြစ်ပါတယ်။ အခပေးရတဲ့ Premium သင်ခန်းစာတွေရှိသလို၊ အခမဲ့ရတဲ့ သင်ခန်းစာတွေလည်း ရှိပါတယ်။ ရှင်းလင်းချက်တွေကလည်း လွယ်ကူနားလည်လွယ်တဲ့အတွက် Laracasts က သင်ခန်းစာတွေကိုလည်း ဆက်လက်လေ့လာသွားဖို့ အကြံပြုပါတယ်။

- <https://laracasts.com/>

နိဂုံးချုပ်

ဒီစာအုပ်ဟာ ကူးစက်မြန်ကပ်ရောဂါတစ်ခုဖြစ်တဲ့ COVID-19 ကိုရိုနာဗိုင်းရပ် ဖြစ်ပွားနေချိန်၊ ရောဂါကူးစက်မှု ကာကွယ်တားဆီးရေးအတွက် တစ်ကမ္ဘာလုံးအတိုင်းအတာနဲ့ Social Distancing နဲ့ Work From Home လှုပ်ရှားမှုကို ဆောင်ရွက်နေစဉ်ကာလ၊ အလုပ်နဲ့ သင်တန်းကျောင်းကို ခဏပိတ်ထားပြီး အိမ်ထဲမှာပဲ နေစဉ်မှာ ရေးဖြစ်ခဲ့တဲ့ စာအုပ်နှစ်အုပ်ထဲက တစ်အုပ်ပါ။ မှတ်မှတ်ရရပါပဲ။

တစ်အုပ်က **React - လို - တို - ရှင်း** ဖြစ်ပြီး ဒီ **Laravel - လို - တို - ရှင်း** က နောက်တစ်အုပ်ဖြစ်ပါတယ်။ ဒီစာအုပ်ကို ဖတ်ရှုလေ့လာလို့ အဆင်ပြေတယ်ဆိုရင် **React - လို - တို - ရှင်း** စာအုပ်ကိုလည်း ဆက်လက်လေ့လာဖို့ အကြံပြုပါတယ်။ အခုလိုပဲ အခြေခံကျကျနဲ့ လိုရင်းတိုရှင်း ရေးသားထားတာပါ။

ဒီစာအုပ်ကို စတင်ရေးသားတော့မယ်လို့ စီစဉ်ကတည်းက Pre-Order စနစ်နဲ့ ဝိုင်းဝန်းမှာယူကြသူများအားလုံးကိုလဲ ဒီနေရာကနေပဲ ကျေးဇူးတင်ကြောင်း ပြောလိုပါတယ်။ ဒီလိုအားပေးပံ့ပိုးမှုတွေ ရှိခဲ့လို့သာ ကြိုးစားပြီး ရေးဖြစ်ခဲ့တာပါ။ အားလုံးပဲ ကပ်ဘေးတွေကို ကျော်လွှားနိုင်ပြီး ကိုယ်စိတ်နှစ်ဖြာ ကျန်းမာချမ်းသာ ကြပါစေလို့ ဆုတောင်းလိုက်ပါတယ်။

အိမောင် (Fairway)

၂၀၂၀ ပြည့်နှစ်၊ မေလ (၁၀) ရက်နေ့တွင် ရေးသားပြီးစီးသည်။