

# آموزش فارسی لاراول ۵

(Laravel 5)

## آموزش فارسی

# AmozeshFarsi.ir

فهرست

۵	آموزش نصب لاراول ۵
۶	پیکربندی لاراول ۵
۸	دسترسی به مقادیر پیکر بندی :
۹	پوشه بندی جدید لاراول ۵
۱۰	روتینگ <b>ROUTING</b> در لاراول ۵
۱۲	مسیر با پارامتر
۱۲	افزودن عبارت منظم به پارامترها
۱۳	مسیردهی به یک کنترلر و اکشن
۱۳	نامگذاری مسیر
۱۴	مسیردهی گروهی
۱۴	کنترلر ها در لاراول ۵
۱۵	کنترلرها و فضای نام(namespace)
۱۶	استفاده از middleware در کنترلر
۱۷	کنترلرهای RESTful
۱۸	کار با <b>view</b> در لاراول ۵
۱۹	بررسی وجود فایل <b>view</b>
۱۹	رندر کردن <b>view</b> از طریق مسیر فایل
۲۱	درخواست های <b>HTTP</b> در لاراول ۵
۲۳	کوکی ها
۲۳	فایل ها
۲۴	پاسخ های <b>HTTP</b> در لاراول ۵
۲۴	پاسخ ساده
۲۴	ایجاد پاسخ دلخواه
۲۵	Redirect
۲۶	ایجاد پاسخ به صورت JSON
۲۶	ایجاد پاسخ به صورت دانلود فایل
۲۷	<b>blade</b> در لاراول ۵
۲۷	کار با موتور قالب <b>Blade</b> و ایجاد <b>Layout</b>
۲۷	تعریف یک <b>Layout</b> ساده
۲۸	چاپ داده یا متغیر ها در <b>blade</b>

۲۹	اینکلود کردن فایل view در view دیگر
۳۰	توضیحات در Blade
۳۰	توابع کمکی در لاراول ۵
۳۲	Middleware ها در لاراول ۵
۳۲	تعریف یک middleware
۳۴	Session ها در لاراول ۵
۳۴	ذخیره مقدار در یک سشن
۳۴	افزودن مقدار به یک session آرایه ای
۳۵	داده های فلش
۳۵	ذخیره سشن ها در دیتابیس
۳۶	اعتبار سنجی در لاراول ۵
۳۹	نمایش پیغام های خطا در view
۳۹	ایجاد یک قانون اعتبار سنجی دلخواه
۳۹	ایجاد پیغام خطای دلخواه برای قوانین اعتبارسنجی
۴۱	کار با دیتابیس در لاراول ۵
۴۱	مباحث پایه کار با دیتابیس
۴۱	اجرای کوئری با کلاس DB
۴۲	کار با دیتابیس با Query Builder
۴۳	استفاده از OR یا AND برای جدا کردن شرط ها
۴۳	استفاده از متدهای جادویی شرط
۴۴	استفاده از Order By و Group By و Having با کوئری بیلدر
۴۴	JOIN کردن
۴۵	درج کردن (INSERT)
۴۵	به روزرسانی (UPDATE)
۴۶	حذف کردن (Delete)
۴۷	قفل کردن جدول هنگام اجرای عملیات
۴۷	Eloquent در لاراول ۵
۴۸	درج کردن با Eloquent
۴۹	به روزرسانی رکوردها
۴۹	حذف رکوردها
۵۰	Relationships جداول در لاراول ۵

۵۰	ارتباطات (Relationships)
۵۰	ارتباط One To Many
۵۱	ارتباط Many To Many
۵۲	درج کردن در جدول رابطه دار
۵۳	صفحه بندی در لارا اول ۵
۵۵	Migration در لارا اول ۵
۵۵	ایجاد یک migration
۶۰	Hash در لارا اول ۵
۶۱	Authentication در لارا اول ۵
۶۳	Authentication به کاربر خاص:
۶۳	اعتبار سنجی کاربر بدون عمل لاگین:
۶۴	لاگین کردن کاربر فقط برای یک درخواست:
۶۴	Authentication Events
۶۴	آپلود فایل در لارا اول ۵
۶۷	ارسال ایمیل در لارا اول ۵
۷۰	Reset Password در لارا اول ۵
۷۴	افزودن کلاس و پکیج در لارا اول ۵

## آموزش نصب لاراو ۵

خب قبل از اینکه بخواهید فریم ورک لاراو ۵ رو نصب کنید باید مطمئن باشید که extension های زیر روی سرورتان نصب باشد و ورژن PHP سرور هم باید ۵.۴ یا بیشتر باشد

- Mcrypt
- OpenSSL
- Mbstring
- Tokenizer

برای اطلاع از فعال بودن این extension ها و همچنین نسخه php روی سیستم می تونید با استفاده از دستور `phpinfo()` به این اطلاعات دست پیدا کنید و در صورت عدم نصب هر کدام با توجه به سیستم عاملتون اقدام به نصب و فعال کردن آنها بکنید. (البته اگر قرار باشه لاراو را روی کامپیوتر خودتون نصب کنید نیازی به این حساسیت ها نیست و اگر extension ای نصب نبود میتونید با استفاده از فایل `php.ini` اون رو فعال کنید.)

بهترین راه نصب لاراو ۵ استفاده از `composer` است که در صورت نصب نبودن روی سیستم تان می توانید از اینجا آن را دریافت و نصب کنید. (البته این سایت معادل فارسی شده هم داره. برای ورود به آن اینجا کلیک نمایید)

ترمینال رو توی لینوکس یا `cmd` رو توی ویندوز باز کنید و ابتدا به دایرکتوری که میخواید فریمورک رو داخلش نصب کنید (پوشه `root` نرم افزار شبیه ساز سروتان مثل `xampp` یا `lamp` و یا `wamp`) بروید مثلا با یکی از دستورات زیر که البته ممکن است مکان پوشه `root` در سیستم شما متفاوت باشد:

کد:

```
// for linux ubuntu
cd /var/www/html
//for windows and xampp
cd c:\xampp\htdocs
//for windows and wamp
cd c:\wamp\www
```

حالا می تونید با تایپ دستور زیر توی ترمینال آخرین نسخه لاراوول رو دانلود و نصب کنید که یک پوشه به نام laravel ساخته میشود:

کد:

```
composer create-project laravel/laravel --prefer-dist
```

نکته: افرادی که از لینوکس استفاده می کنند باید به پوشه های storage و vendor مجوز نوشتن فایل رو بهش بدهید.

در صورتی که composer در سیستم شما نصب نمی شود یا مشکلی دارد میتونید فایل های فریمورک لاراوول را از آدرس زیر دریافت و در مسیر پوشه root سرورتان extract کنید:

<http://fian.my.id/larapack/>

## پیگر بندی لاراوول ۵

توی پوشه اصلی لاراوول یک فایل به نام env وجود دارد که می توانید تنظیمات برنامه تان و دیتابیس پروژه را در اینجا تعیین کنید:

کد:

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=zGKCjTPbzET3WiHhKCxSpTBNCuUVWwLc
DB_HOST=localhost
DB_DATABASE=learninglaravel
DB_USERNAME=root
DB_PASSWORD=secret
```

به طور مثال اگر APP\_DEBUG را روی true ست کنید خطاهای برنامه نویسی در هنگام کدنویسی برایتان قابل مشاهده خواهد بود و مناسب برای حالت development هست و در هنگام آپلود سایت روی هاست آن را false قرار دهید.

بهرتره مقدار APP\_KEY را هم با تایپ دستور زیر در ترمینال تغییر دهیم:

کد:

```
php artisan key:generate
```

سایر تنظیمات رو هم میتونید در پوشه config در فایل مورد نظرش اعمال کنید. به طور مثال می توانید در فایل app.php مقدار timezone رو به Asia/Tehran تغییر دهید.

## توضیحات بیشتر در مورد پیکربندی لارا اول

شما می توانید داخل فایل app.php در پوشه config تنظیمات برنامه را اعمال کنید. تنظیمات به صورت یک جفت کلید/مقدار هستند. بعضی از آیتم ها مقدار خودشان را توسط تابع کمی env از فایل env واقع در دایرکتوری root پروژه که در پست قبلی توضیح دادم می گیرند به طور مثال:

کد پی اچ پی:

```
'debug' => env('APP_DEBUG'),
'key' => env('APP_KEY', 'SomeRandomString'),
```

debug و key مقدار خودش رو از فایل env می گیرند در صورتی که در فایل env. برایشان مقداری ست نکرده باشیم می توانیم به تابع env() پارامتر دومی بدهیم که نشانگر مقدار آن هست. در مثال بالا key به این صورت است و اگر در فایل env. آن را حذف کنیم از این مقدار پیش فرض استفاده خواهد کرد.

## در زیر توضیح مختصری برای هر آیتم آن میدهم:

debug: اگر مقدار آن را true ست کنید برنامه در مد development خواهد بود و خطاهای برنامه نشان داده می شود و اگر false باشد در مد production می باشد و مناسب برای publish و استفاده نهایی برنامه هست.

- url: آدرس url پروژه را در اینجا ست میکنیم مثلا `http://localhost/laravel/public`
- timezone: موقعیت زمانی را مشخص می کنیم که برای مثال در کشور ایران Asia/Tehran ست می کنیم.
- locale: در مسیر `resources/lang` می توانیم یک پوشه دیگر به نام `fa` ایجاد کرده تا در آن پیغام ها و متون فارسی را تایپ کنیم تا در برنامه از آنها استفاده کنیم. به طور مثال یک کاربرد آن در فارسی سازی پیغام های اعتبارسنجی فرم ها می باشد. مقدار این آیتم را `fa` که همنام آن پوشه که ایجاد کردیم ست میکنیم.
- fallback\_locale: در صورتی که locale موردنظر برای آن رشته موجود نبود از این locale استفاده شود.
- key: کلید برنامه که یک رشته تصادفی هست و در رمزنگاری های برنامه توسط لارا اول مورد استفاده قرار می گیرد. نحوه ست کردن آن را در پست قبلی توضیح دادم.

سایر موارد را در جای مناسب خودش توضیح خواهم داد.

لاراول ۵ به طور پیش فرض از دایرکتوری `app` تحت `namespace` ای به نام `App` استفاده میکند که هنگام ایجاد کلاس هایتان از آن استفاده میکنید که شما می توانید با استفاده از دستور زیر و تایپ در ترمینال آن فضای نام را به نام دلخواهتان تغییر دهید مثلا در مثال زیر من آن را به `Hamo` تغییر دادم:

کد:

```
php artisan app:name Hamo
```

بعد از اجرای این دستور لاراول به طور خودکار تمام `namespace` های استفاده شده در کلاس هایتان را به نام جدید تغییر خواهد داد.

## دسترسی به مقادیر پیکر بندی:

با استفاده از کلاس `Config` هم می توانید مقادیر `config` رو با استفاده از متد `get` بدست بیارید یا مقدار جدیدی را با استفاده از متد `set` ست کنید به مثال های زیر توجه کنید:

کد پی ایچ پی:

```
$value = Config::get('app.timezone');  
Config::set('app.timezone', 'Asia/Tehran');
```

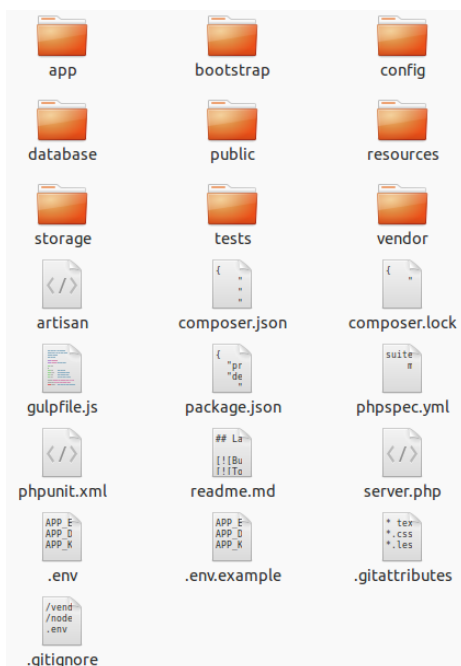
همچنین می توانید از تابع کمکی `config` هم استفاده کنید:

کد پی ایچ پی:

```
$value = config('app.timezone');
```



## پوشه بندی جدید لاراول 5



در بالا تصویری از دایرکتوری root لاراول قرار دادم. در زیر درمورد آنها توضیحاتی می دهیم :

- **app** : این دایرکتوری حاوی تمام کدهای برنامه تان از جمله کنترلرها و مدل های برنامه تان هست. با این دایرکتوری زیاد سروکار خواهیم داشت.
- **bootstrap**: این دایرکتوری حاوی یک سری فایل برای autoloading و راه اندازی فریمورک هست.
- **config**: حاوی تمام فایل های پیکربندی برنامه تان است.
- **database** : حاوی فایل های migration و seed است.
- **public** : فایل های استاتیک و front-end برنامه تان از قبیل javascript , css, images در اینجا قرار میگیرند.
- **resources**: در این دایرکتوری فایل های view برنامه و فایل های locale و زبان در آن قرار می گیرند.
- **storage**: در این دایرکتوری فایل هایی که توسط موتور پوسته blade کامپایل می شوند و همچنین مکان ذخیره سازی فایل های سشن و کش و سایر فایل هایی که توسط فریمورک ایجاد می شوند می باشد.
- **test**: حاوی فایل های تست خود کار برنامه است.
- **vendor**: حاوی تمام third-party ها و وابستگی هایی که توسط composer به برنامه اضافه می شوند هست.

داخل دایرکتوری app می توانید مدل ها را ایجاد کنید و همچنین در مسیر app/Http/controllers می توانید کنترلرهای برنامه را ایجاد کنیم و همچنین فایل routes.php که در مسیر app/Http قرار دارد که مدیریت مسیره‌ها از آن استفاده میکنیم از جمله فایل ها و دایرکتوری های پر کاربرد ما در این فریمورک هستند.

فایل های view برنامه را هم در مسیر resources/views قرار می دهیم. در قسمت های بعدی نحوه مسیره‌دهی و ایجاد کنترلر و ویوها را خواهیم آموخت.

## روتینگ ROUTING در لارا اول 5

از مزیت های فریم ورک لارا اول نسبت به سایر فریمورک های PHP مبحث Routing آن است که می توان مدیریت خوبی روی مسیره‌ها داشت. در مسیر app/Http و فایل routes.php می توانیم تمامی مسیره‌های برنامه را در آنجا تعریف و مدیریت کنیم. این فایل توسط کلاس App\Providers\RouteServiceProvider بارگزاری میشود.

کد پی اچ پی:

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

کلاس Route چند متد دارد که نوع درخواست http را مشخص میکند. در مثال بالا متد get فقط در خواست های GET به این مسیر را قبول میکند. سایر متدها که نوع درخواست http را مشخص میکنند , post , put, patch, delete می باشند. این متد دوتا پارامتر می گیرد که اولی مسیری است که بعد از نام دامنه سایت می آید مثلا در آدرس http://www.example.com/about مسیری که وارد میکنیم about است.

در پارامتر دومی هم می توانیم بدون استفاده از کنترلر و اکشن و با دادن یک تابع بی نام در همین روتر آن را مدیریت کنیم.

کلاس Route دارای متد دیگری به نام match هست که می توانیم چند نوع درخواست http را به یک مسیر مجاز کنیم در مثال زیر مسیر هردونوع درخواست GET و POST را قبول می کند:

کد پی اچ پی:

```
Route::match(['get', 'post'], '/', function()
{
    return 'Hello World';
});
```

در صورتی که بخواهیم مسیر همه در خواست ها را قبول کنید از متد any استفاده میکنیم مثلا آدرس http://www.example.com/foo هر درخواستی را قبول میکند:

کد پی اچ پی:

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

HTMLدرخواست های PUT , DELETE یا PATCH را پشتیبانی نمی کند برای اینکه یک فرم HTML را با این متدها تعریف کنیم کافیه یک تگ input از نوع hidden و با نام \_method تعریف میکنیم و به value آن یکی از مقادیر PUT, DELETE, PATCH را بدهید مثلا:

کد پی اچ پی:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
</form>
```

در مثال بالا آدرس http://www.example.com/foo/bar در روتر با متد put قابل دریافت است که می توانیم برای DELETE , PATCH هم به همین صورت عمل کنیم. کاربرد این متدها را در بخش کنترلر ها تشریح خواهیم کرد. همچنین یک تگ از نوع مخفی به نام \_token هم در فرم وجود دارد که در یک پست جداگانه در مورد فرم ها و کار با آنها توضیح خواهیم داد.

## مسیر با پارامتر

به همراه مسیر می توانیم هر تعداد پارامتر را هم ارسال کنیم فقط کافی است نام پارامترها را داخل آکولاد قرار دهیم. به مثال های زیر توجه کنید :

کد پی اچ پی:

```
Route::get('user/{id}', function($id)
{
    return 'User '.$id;
});

Route::get('user/{name?}', function($name = null)
{
    return $name;
});

Route::get('user/{name?}', function($name = 'John')
{
    return $name;
});
```

در مثال های بالا همانطور که مشاهده کردید می توانیم برای پارامترها یک مقدار پیش فرض یا null هم در نظر گرفت تا در صورت وارد نکردن مقداری برای پارامتر در url خطایی ایجاد نشود. همچنین باید جلوی نام پارامتر های اختیاری یک علامت ؟ قرار دهیم.

## افزودن عبارت منظم به پارامترها

می توانیم با افزودن متد where به انتهای متد get برای هر پارامتر یک عبارت منظم هم تعریف کرد تا مثلاً id فقط مقدار عدد مورد قبول باشد. در صورتی که چند پارامتر را بخواهیم برایش عبارت منظم تعریف کنیم آنها را داخل آرایه قرار می دهیم.

کد پی اچ پی:

```
Route::get('user/{id}/{name}', function($id, $name)
{
    //
});
->where(['id' => '[0-9]+', 'name' => '[a-z]+'])
```

همچنین می توانیم برای یک پارامتر خاص در کل برنامه یک عبارت منظم عمومی تعریف کنیم به این صورت که در کلاس RouteServiceProvider در دایرکتوری app/Providers در متد boot این عبارت را

قرار دهیم مثلا در مثال زیر کاربر در routing هرجایی از پارامتر id استفاده کرد فقط مجاز به دادن مقدار عددی به آن است و دیگر مانند بالا نیاز به تعریف متد where نیست:

کد پی اچ پی:

```
$router->pattern('id', '[0-9]+');
```

## مسیردهی به یک کنترلر و اکشن

کد پی اچ پی:

```
Route::get('user/{id}', 'UserController@showProfile');
```

در پارامتر دوم فقط کافی است بین نام کلاس کنترلر و اکشن یک علامت @ قرار دهیم.

## نامگذاری مسیر

با استفاده از کلمه as می توانیم برای مسیر یک نام هم تعریف کنیم و همچنین با استفاده از uses می توانیم آن را به اکشن و کنترلر خاصی هدایت کنیم.

کد پی اچ پی:

```
Route::get('user/profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

از کاربردهای نامگذاری مسیر برای ایجاد و ساختن url است که می توانیم با استفاده از تابع کمکی route نام مسیر را به آن بدهیم مثلا در مثال بالا با دادن نام profile آدرس:  
<http://www.example.com/user/profile>  
 ایجاد خواهد شد و همچنین برای ریدایرکت به یک مسیر هم کاربرد دارد.

کد پی اچ پی:

```
$url = route('profile');

$redirect = redirect()->route('profile');
```

## مسیردهی گروهی

در لارا اول می توانیم یک دسته از مسیرها را که مثلا در یک قسمت از url خود مشترک هستند یا middleware مشترکی دارند و یا دارای یک namespace مشترک هستند را در یک گروه قرار دهیم. همچنین می توانیم sub-domain ها را از این طریق مدیریت کنیم.

کد پی اچ پی:

```
Route::group(['prefix' => 'admin'], function()
{
    Route::get('users', function()
    {
        // Matches The "/admin/users" URL
    });
});
```

در مثال بالا تمامی مسیرهایی که با admin شروع می شوند را داخل این گروه قرار می دهیم.

## کنترلرها در لارا اول ۵

یکی از سه عنصر اصلی الگوی طراحی MVC کنترلرها هستند. در فایل routing.php می توانیم درخواست ها را به یک کنترلر و اکشن خاصی ارسال کنیم به طور مثال آدرس <http://www.example.com/user/5> را در مثال زیر به کنترلر UserController و اکشن showProfile هدایت می کند.

کد پی اچ پی:

```
Route::get('user/{id}', 'UserController@showProfile');
```

تعریف کنترلر : کنترلرها در مسیر دایرکتوری `app/Http/Controllers` قرار می‌گیرند .

کد پی‌اچ‌پی:

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

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

## کنترلرها و فضای نام (namespace)

برای هر کلاس باید namespace آن را تعریف کنیم که این فضای نام در واقع مسیر قرارگیری کلاس از پوشه `app` می‌باشد و برای کنترلرها `App\Http\Controllers` تعریف می‌کنیم. در صورتی که داخل دایرکتوری `Controllers` یک دایرکتوری دیگر مثلاً به نام `Auth` ایجاد کرده باشیم و کنترلری در آن تعریف کنیم فضای نام به صورت `namespace App\Http\Controllers\Auth` می‌باشد.

نکته : همیشه نام کلاس‌های کنترلر را به صورت `PascalCase` و در انتهای آن کلمه `Controller` را بیاورید. بهتر است اکشن‌ها را هم به صورت `camelCase` نامگذاری کنید. البته من خودم همیشه عادت دارم کلاس‌های کنترلر و مدل را با ترمینال ایجاد کنم که شما هم می‌توانید با این دستور یک کنترلر بدون هیچ متدی ایجاد کنید:

کد:

```
php artisan make:controller UserController --plain
```

## استفاده از middleware در کنترلر

همانطور که در پست قبلی توضیح دادم می توانیم برای هر مسیر خاص یک کلاس میان افزار اضافه کنیم تا درخواست ها فیلتر شوند. مثلا در مثال زیر برای مسیر میان افزار auth را اضافه کردیم.

کد پی اچ پی:

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

در مثال زیر همانطور که مشاهده می کنید سه مثال از استفاده از میان افزار در کنترلرها را آورده است که در متد سازنده کلاس هم قرار می گیرند:

کد پی اچ پی:

```
class UserController extends Controller {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }
}
```

در مثال دوم میان افزار را با استفاده از کلمه **only** فقط به اکشن های **fooAction** و **barAction** محدود کردیم و فیلتر فقط به این اکشن ها اعمال شود و در مثال سوم با استفاده از کلمه **except** میان افزار به همه اکشن ها اعمال شود به جز اکشن های **fooAction** و **barAction**. در لارا اول همچنین می توانیم به مسیرهی به یک اکشن را به صورتی ساده تر هم انجام دهیم مثلا با تعریف مسیره به این صورت:

```
Route::controller('users', 'UserController');
```



با افزودن درخواست http به ابتدای نام اکشن با توجه به نوع درخواست به اکشن مورد نظر تحویل داده می شود:

کد پی اچ پی:

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

**نکته:** اگر می خواهید برخی از مسیرها را نامگذاری کنید کفایت پارامتر سومی هم به صورت آرایه در نظر بگیرید و کلید آرایه نام اکشن و مقدار آن نام مسیر باشد:

کد پی اچ پی:

```
Route::controller('users', 'UserController', [
    'anyLogin' => 'user.login',
]);
```

## کنترلرهای RESTful

در لاراوول می توانیم با دستور زیر در ترمینال کنترلرهایی با اکشن های خاصی ایجاد کنیم که هر اکشن یک مسیر و درخواست http را تحویل میگیرند. به طور مثال کنترلر PhotoController را ایجاد می کنیم:

کد:

```
php artisan make:controller PhotoController
```

مسیر را هم به این صورت در فایل routes.php تعریف می کنیم:

کد پی اچ پی:

```
Route::resource('photo', 'PhotoController');
```

حالا اگر url را به صورت <http://www.example.com/photo> بنویسیم اکشن index درخواست را دریافت میکند. در تصویر زیر می توانید اطلاعات کاملی را از تمام اکشن ها داشته باشید **verb**. نوع درخواست **http** و **path** مسیری که در url وارد میکنیم و **action** اکشنی که این درخواست را دریافت میکند و **route name** هم نام مسیر می باشد.

همچنین می توانیم فقط اکشن های خاصی را به صورت RESTful تعریف کنیم:

کد پی اچ پی:

```
Route::resource('photo', 'PhotoController',
    ['only' => ['index', 'show']]);

Route::resource('photo', 'PhotoController',
    ['except' => ['create', 'store', 'update', 'destroy']]);
```

## کار با view در لاراوول ۵

در لاراوول ۵ view ها را در مسیر `resources/views` قرار می دهیم. شما می توانید آنها را با استفاده از موتور قالب `Blade` و یا به صورت معمولی ایجاد کنید. در مثال زیر فایل `greeting.php` را در مسیر ذکر شده قرار می دهیم و در آن دستورات زیر را قرار میدهیم:

کد:

```
<!-- View stored in resources/views/greeting.php -->
<html>
  <body>
    <h1>Hello, <?php echo $name; ?></h1>
  </body>
</html>
```

با استفاده از تابع کمکی `view` هم می توانیم فایل ویو را `render` کنیم. این تابع دو پارامتر می گیرد که اولی نام فایل ویو موردنظر بدون قرار دادن فرمت آن و دومین پارامتر آرایه ای از داده هایی هست که به فایل ویو می فرستیم. کلید آرایه در فایل ویو به صورت نام متغیر قابل استفاده است. در مثال زیر کاربر با وارد کردن آدرس <http://www.example.com> به او `Hello, James` نمایش داده می شود.

کد پی اچ پی:

```
Route::get('/', function()
{
    return view('greeting', ['name' => 'James']);
});
```

در صورتی که فایل view داخل یک دایرکتوری باشد کافی است نام دایرکتوری و فایل را با یک نقطه از هم جدا کنید:

کد پی اچ پی:

```
return view('admin.profile', $data);
```

در مثال فوق فایل ویو در مسیر resources/views/admin/profile.php قرار دارد.

همچنین به روش های زیر هم می توانیم داده را به ویو ارسال کنیم:

کد پی اچ پی:

```
// Using conventional approach
$view = view('greeting')->with('name', 'Victoria');

// Using Magic Methods
$view = view('greeting')->withName('Victoria');
```

متد with دو پارامتر میگیرد که اولی نام متغیر و دومی مقدار آن هست. همچنین می توانید به روش دوم که در انتهای متد with نام متغیر را اضافه و مقدارش را به عنوان پارامتر به آن می دهیم.

## بررسی وجود فایل view

کد پی اچ پی:

```
if (view()->exists('emails.customer'))
{
    //
}
```

## رندر کردن view از طریق مسیر فایل

کد پی اچ پی:

```
Route::get('/', function(){
    return view()-
>file('/var/www/html/laravel/public/greeting.php', ['name' => 'James']);
});
```

همانطور که می بینید کاربرد آن برای مواقعی است که شما فایل **view** که خارج از مسیر **resources/views** تعریف کرده اید را بتوانید رندر کنید. در مثال بالا من فایل ویو را در پوشه **public** ایجاد کردم.

## درخواست های HTTP در لاراول 5

در فریم ورک لاراول درخواست های http که با متدهای GET , POST , ... ارسال می کنیم را می توانیم مقادیر آنها را با استفاده از کلاس Request دریافت کنیم:

کد پی اچ پی:

```
$name = Request::input('name');
```

نکته: برای استفاده از هر کلاسی در کلاس های کنترلر ابتدا باید آن کلاس را با استفاده از دستور use ایمپورت کنیم. در مثال بالا هم بایستی به این صورت قبل از تعریف کلاس کنترلر مورد نظر کلاس Request را ایمپورت کنیم.

کد پی اچ پی:

```
use Request;
```

همچنین می توانیم به روش دیگری هم مقادیر را به دست بیاوریم. به این صورت که ابتدا کلاس Illuminate\Http\Request را به کنترلر مورد نظر ایمپورت می کنیم سپس دستور Request \$request را به عنوان پارامتر به اکشن مورد نظر می دهیم. در طول برنامه داخل اکشن می توانیم از متغیر \$request استفاده کنیم.

کد پی اچ پی:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {
    /**
     * Store a new user.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name'); //
    }
}
```

می توانیم برای یک ورودی مقداری پیش فرض هم تعیین کنیم تا در صورتی که مقداری برای آن ست نشده بود این مقدار جایگزین آن شود:

کد پی ایچ پی:

```
$name = Request::input('name', 'Sally');
```

با استفاده از متد `has` می توانیم بررسی کنیم که آیا ورودی با این مقدار وجود دارد یا خیر:

کد پی ایچ پی:

```
if (Request::has('name'))
{
    //
}
```

با استفاده از متد `all` می توانیم تمامی ورودی ها را دریافت کنیم.

کد پی ایچ پی:

```
$input = Request::all();
```

همچنین می توانیم فقط برخی ورودی ها یا همه ورودی ها به جز برخی را دریافت کنیم.

کد پی ایچ پی:

```
$input = Request::only('username', 'password');
$input = Request::except('credit_card');
```

هنگامی که مقدار ورودی یک آرایه باشد می توانیم با استفاده از نقطه به مقدار آیتم مورد نظر دست پیدا کرد:

کد پی ایچ پی:

```
$input = Request::input('products.0.name');
```

همچنین می توانیم به مقادیر `flash` که توسط سشن ایجاد می شوند و به صورتی هستند که فقط برای درخواست بعدی معتبر هستند و از بین می روند هم به صورت های زیر دسترسی داشته باشیم:

کد پی ایچ پی:

```
Request::flash();
Request::flashOnly('username', 'email');
Request::flashExcept('password');
```

در مثال دوم و سوم هم مثل قبل که دیدیم فقط یا به جز برخی موارد دسترسی داریم.

می توانیم مقادیر ورودی ها را دوباره با استفاده flash به صفحه قبلی یا صفحه دیگری ارسال کنیم:

کد پی اچ پی:

```
return redirect('form')->withInput();
return redirect('form')->withInput(Request::except('password'));
```

کاربرد آن در فرم ها می باشد که اگر بعد از اعتبارسنجی ورودی ها دارای خطایی باشد و بخواهیم دوباره به صفحه فرم بازگردیم ورودی های فرم که کاربر نوشته از بین نروند. در مثال دوم به password اجازه حفظ شدن ندادیم.

برای چاپ مقادیر قبلی هم باید داخل تکست باکس های فرم مقدارش را به این صورت چاپ کنیم:

کد پی اچ پی:

```
<input type="text" name="email" value="<?php echo old('name') ?>">
```

## کوکی ها

می توانیم به مقدار یک کوکی هم به این صورت دسترسی داشته باشیم:

کد پی اچ پی:

```
$value = Request::cookie('name');
```

## فایل ها

فایلی که آپلود شده را می توانیم به این صورت اطلاعاتش دریافت کنیم. در مثال زیر نام فیلد فایل در فرم photo بوده است:

کد پی اچ پی:

```
$file = Request::file('photo');
```

در مثال زیر بررسی می کند که آیا این فایل با این نام وجود دارد:

کد پی اچ پی:

```
if (Request::hasFile('photo'))
{
    //
}
```

مقداری که متد file در کلاس Request به ما می دهد یک آبجکت از کلاس `Symfony\Component\HttpFoundation\File\UploadedFile` که می توانید با متدهای آن برای کار با فایل کار کنید.

کد پی ایچ پی:

```
if (Request::file('photo')->isValid())
{
    //
}
```

در مثال بالا بررسی می کند که آیا فایل آپلود شده صحیح و بدون خطا می باشد: با استفاده از متد move می توانیم فایل را به مسیر مورد نظر که به عنوان پارامتر اول به آن می دهیم و همچنین نام فایل که اختیاری است ذخیره کنیم.

کد پی ایچ پی:

```
Request::file('photo')->move($destinationPath);
Request::file('photo')->move($destinationPath, $fileName);
```

## پاسخ های HTTP در لاراول ۵

### پاسخ ساده

بعد از دریافت درخواست در لاراول و انجام عملیات مورد نظر باید پاسخی هم ایجاد کنیم. ساده ترین نوع پاسخ return رشته هست که قبلا هم با آن آشنا شدیم:

کد پی ایچ پی:

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

### ایجاد پاسخ دلخواه

با استفاده از کلاس Response یا تابع کمکی response می توانیم یک پاسخ دلخواه ایجاد کنیم مثلا مثال زیر را در نظر بگیرید:

کد پی ایچ پی:

```
return response($content, $status)
->header('Content-Type', $value);
```



محتویات را به عنوان پارامتر اول و `[url=http://en.wikipedia.org/wiki/List_of_HTTP_status_codes][url]` status code را به عنوان پارامتر دوم به آن بدهیم و همچنین با استفاده از متد `header` نوع هدر را هم مشخص کنیم مثلا `application/pdf`. همینطور که در مثال زیر می بینید یک فایل ویو و همچنین یک فایل کوکی را هم به عنوان پاسخ ارسال کنید و استفاده از متدها به صورت زنجیره ای امکان پذیر است.

کد پی اچ پی:

```
return response()->view('hello')->header('Content-Type', $type)
->withCookie(cookie('name', 'value'));
```

## Redirect

با استفاده از تابع کمکی `redirect` و افزودن مسیر به آن می توانیم به مسیر مورد نظر هدایت شویم.

کد پی اچ پی:

```
return redirect('user/login');

return redirect('user/login')->with('message', 'Login Failed');
```

همچنین می توانیم به همراه ریدایرکت کردن یک داده `flash` هم ارسال کنیم. با استفاده از متد `back` می توانیم به مسیر قبلی که بودیم دوباره هدایت شویم.

کد پی اچ پی:

```
return redirect()->back();

return redirect()->back()->withInput();
```

در مثال دومی می توانیم درخواست هایی که به این مسیر آمده را هم دوباره به مسیر قبلی ارسال کنیم که در پست قبلی نحوه کار با آنها را مشاهده کردیم.

می توانیم با استفاده از نام مسیر که در فایل `routes.php` تعریف میکنیم هم ریدایرکت را با استفاده از متد `route` انجام دهیم.

کد پی اچ پی:

```
return redirect()->route('login');

// For a route with the following URI: profile/{id}

return redirect()->route('profile', [1]);
```

همچنین می توانیم با استفاده از یک آرایه به عنوان پارامتر دوم متد `route` داده هم به آن ارسال کنیم. می توانیم با استفاده از متد `action` به یک اکشن در کلاس کنترلر دیگری هدایت شویم که بایستی نام کلاس با فضای نام آن نوشته شود و همچنین در صورت وجود پارامتر به صورت آرایه به عنوان پارامتر دوم به آن اضافه می کنیم.

کد پی اچ پی:

```
return redirect()->action('App\Http\Controllers\HomeController@index ');

return redirect()-
>action('App\Http\Controllers\UserController@profile', ['user' => 1]);
```

## ایجاد پاسخ به صورت JSON

با استفاده از متد `json` که یک آرایه را به عنوان پارامتر ورودی دریافت می کنید و خروجی آن به صورت JSON می باشد.

کد پی اچ پی:

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

## ایجاد پاسخ به صورت دانلود فایل

با استفاده از متد `download` که مسیر فایل را به عنوان پارامتر می گیرد و دو پارامتر اختیاری دیگر که نام فایل و هدرهای فایل هست را دریافت میکند.

کد پی اچ پی:

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

## blade در لارا اول ۵

### کار با موتور قالب Blade و ایجاد Layout

در فریم ورک لارا اول برای ایجاد view ها میتونید از موتور قالب Blade هم استفاده کنید که کارتون رو در ایجاد layout ها و کدنویسی خیلی آسون میکنه. شما می تونید بخش هایی از وبسایت از جمله هدر و فوتر و منوها و ... که در تمام صفحات وبسایت یکی هستن را داخل یک فایل layout ایجاد کرده و در فایل های دیگر قابل ارث بردن هست. این فایل ها با فرمت blade.php ایجاد می شوند.

### تعریف یک Layout ساده

در مسیر resources/views یک پوشه به نام layouts ایجاد کرده و فایل master.blade.php را داخل آن ایجاد کرده و کدهای زیر را داخل آن می نویسیم:

کد پی اچ پی:

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

اکثر دستورات blade با علامت @ شروع می شوند. با استفاده از دستور yield می توانیم یک بخش را ایجاد کنیم که بعدا در فایل هایی که از آن ارث برده می شوند بتوانید محتوایی که در هر فایل متفاوت است را در آن قرار دهیم. نحوه استفاده از layout بالا را در فایلی دیگر مشاهده کنید:

## کد پی اچ پی:

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

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@stop

@section('content')
    <p>This is my body content.</p>
@stop
```

همانطور که مشاهده کردید با استفاده از دستور extends می توانید فایل layout را به صفحه اضافه کنید. نحوه آدرس دهی هم به این صورت است که بین دایرکتوری و نام فایل ویو نقطه قرار می دهیم. با استفاده از دستور section که نام yield مورد نظر را به آن می دهیم می توانیم محتوای جدید را داخل آن قرار دهیم. در پایان هم باید stop را بنویسیم yield. ها در فایل layout هیچ محتوایی ندارند اما اگر بخواهیم بخشی را تعریف کنیم که در فایل layout هم محتوا داشته باشند باید از section استفاده با این تفاوت که در layout باید در انتها show قرار دهیم. بخش ها در فایل به ارث برده شده override می شوند برای اینکه بتوانیم محتوای فایل والد رو هم داشته باشیم کافیست در ابتدا یا انتهای محتوای جدید دستور parent را اضافه کنیم. در مثال بالا بخش sidebar به این صورت است.

برای بخش yield می توانیم یک محتوای پیش فرض هم تعیین کنیم مثلا:

## کد پی اچ پی:

```
@yield('section', 'Default Content')
```

## چاپ داده یا متغیر ها در blade

با استفاده از بلاک های دو آکولاده می توانیم یک متغیر یا عبارت قابل چاپ را در صفحه چاپ کنیم.

## کد پی اچ پی:

```
Hello, {{ $name }}.

The current UNIX timestamp is {{ time() }}.
```

همچنین اگر متغیری با نام مورد نظر ست نشده بود یک مقدار پیش فرض برای چاپ در نظر بگیریم تا باعث ایجاد خطا در صفحه نشود.

کد پی اچ پی:

```
{{ $name or 'Default' }}
```

دو آکولاد در blade تمامی دستورات html را escape میکند مانند دستور htmlentities در php عمل میکند. اگر نخواهیم داده ها escape شوند به این صورت انجام دهید:

کد پی اچ پی:

```
Hello, {!! $name !!}.
```

دستورات شرطی و حلقه ها هم به صورت های زیر قابل نوشتن هستند:

کد پی اچ پی:

```
@if (count($records) === 1)
    I have one record!
@endif
@elseif (count($records) > 1)
    I have multiple records!
@endif
@else
    I don't have any records!
@endif
@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach
```

## اینکود کردن فایل view در view دیگر

مثلا در یک فایل ویو فرم لاگین را طراحی کرده ایم و می خواهیم آن را در چند صفحه استفاده کنیم کافیت آن را مانند مثال زیر در فایل های مورد نظر اینکود کنیم:

کد پی اچ پی:

```
@include('view.name')
@include('view.name', ['some' => 'data'])
```

در مثال بالا view نام پوشه و name نام فایل ویو مورد نظر است. همچنین می توانیم دیتا هم به آن فایل ارسال کنیم.

## توضیحات در Blade

برای نوشتن کامنت یا توضیحات می توانید به صورت زیر عمل کنید:

کد پی اچ پی:

```
{{!-- This comment will not be in the rendered HTML --}}
```

## توابع کمکی در لارا اول ۵

در لارا اول ۵ توابع کمکی یا helper بسیار زیادی در لارا اول وجود دارند که در حین توسعه برنامه به کارتون میان و توی پست های قبلی هم از چندتا ازونا استفاده کردیم مثل تابع view برای کار با آرایه ها و مسیرها و ایجاد url و کار با رشته ها توابع بسیار خوبی دارد. توی این پست میخوام چندتا از پرکاربردهاشو معرفی کنم.

افزودن به آرایه با تابع array\_add

کد پی اچ پی:

```
$array = ['foo' => 'bar'];  
$array = array_add($array, 'key', 'value');
```

تقسیم آرایه به دو آرایه از کلیدها و مقادیر با تابع array\_divide

کد پی اچ پی:

```
$array = ['foo' => 'bar'];  
list($keys, $values) = array_divide($array);
```

گرفتن مسیر فیزیکی دایرکتوری app و public با توابع app\_path و public\_path

کد پی اچ پی:

```
$path = app_path();  
$path = public_path();
```

اجرای دستور htmlentities روی رشته با پشتیبانی از UTF-8 با تابع e

کد پی اچ پی:

```
$entities = e('<html>foo</html>');
```

ایجاد یک رشته تصادفی به طول دلخواه با تابع `str_random` که مثلا مناسب برای ایجاد کلمه عبور است.

کد پی اچ پی:

```
$string = str_random(40);
```

ایجاد مسیر کامل با تابع `url` - پارامتر اولش مسیر نسبی هست و پارامتر دوم هم پارامترهای مسیر در صورت وجود است و پارامتر سوم اگر `true` باشد مسیر با پروتکل `https` ایجاد می شود

کد پی اچ پی:

```
echo url('foo/bar', $parameters = [], $secure = null);
```

ایجاد یک توکن در فرم ها برای جلوگیری از حملات `csrf` با تابع `csrf_token`

کد پی اچ پی:

```
$token = csrf_token();
```

تابع `dd` هم یک متغیر یا آبجکت یا آرایه را می گیرد و به صورتی شبیه `var_dump` نمایش می دهد و برای `debug` کردن خیلی کاربردی هست

کد پی اچ پی:

```
dd($value);
```

## Middleware ها در لاراول ۵

در فریم ورک لاراول middleware ها یک مکانیسم ساده ای را برای فیلتر کردن درخواست های http ورودی به برنامه تان تدارک می بیند. به طور مثال لاراول یک middleware ترجمه فارسیش میشه میان افزار) برای احراز هویت کاربران دارد و در صورتی که کاربری Login نکرده باشد و احراز هویت نشده باشد میان افزار آن را به صفحه لاگین هدایت میکند و گرنه میان افزار به درخواست اجازه ادامه کارش را میدهد. middleware ها در دایرکتوری app/Http/Middleware قرار میگیرند.

### تعریف یک middleware

با تایپ دستور `make:middleware` در ترمینال می توانیم یک میان افزار جدید ایجاد کنیم. در مثال زیر میان افزار `OldMiddleware` را ایجاد کردیم.

#### کد پی اچ پی:

```
php artisan make:middleware OldMiddleware
```

فایل ایجاد شده را باز میکنیم و در متد `handle` شرط زیر را قرار میدهیم به این صورت که درخواست ورودی به نام `age` اگر کوچکتر از ۲۰۰ بود به صفحه `home` ریدایرکت شود و گرنه به درخواست اجازه ادامه کار بدهد.

#### کد پی اچ پی:

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

class OldMiddleware {

    /**
     * Run the request filter.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') < 200)
        {
            return redirect('home');
        }

        return $next($request);
    }
}
```



اکنون برای اینکه بخواهیم از این میان افزار استفاده کنیم ابتدا باید آن را در فایل `app/Http/Kernel.php` ثبت کنیم. اگر می خواهید این میان افزار برای هر درخواست `http` برنامه تان اجرا شود آن را به آرایه `$middleware` اضافه کنید که بعد از این هر درخواستی با این نام را فیلتر خواهد کرد. اگر می خواهید میان افزار فقط به یک مسیر خاص اعمال شود ابتدا باید آن را به آرایه `$routeMiddleware` اضافه کنید به این صورت که کلید آن در آرایه نام خلاصه آن برای استفاده در برنامه به کار می رود:

کد پی ای پی:

```
protected $routeMiddleware = [
    'auth' => 'App\Http\Middleware\Authenticate',
    'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
    'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
    'old' => 'App\Http\Middleware\OldMiddleware',
];
```

حالا می تونید میان افزار را به هر مسیری در فایل `routing.php` مانند مثال های زیر اضافه کنید که دوتا میان افزار `old` و `auth` را به مسیرهای موردنظرمان افزودیم:

کد پی ای پی:

```
Route::post('url/create', ['middleware' => 'old', 'uses'=>'UrlController@create']);

Route::get('admin/profile', ['middleware' => 'auth', function()
{
    //
}]);
```

### Before / After Middleware

همچنین می توانیم میان افزارهای خاصی را ایجاد کنیم که قبل یا بعد از مدیریت درخواست توسط برنامه عملی را اجرا کنند.

## Session ها در لاراول ۵

در لاراول ۵ می توانیم از طریق کلاس Session و هم با استفاده از تابع کمکی session به مقادیر آنها دسترسی داشته باشیم.

### ذخیره مقدار در یک سشن

در مثال زیر با هر دو روش مقداری را در session ذخیره کرده ایم key نام session و value مقدار آن است. برای تعریف چند session کلید و مقدار را داخل یک آرایه قرار دهید.

کد پی ایچ پی:

```
Session::put('key', 'value');
session(['key' => 'value']);
```

باید توجه داشته باشید که برای ست کردن یک session هم در تابع کمکی session باید آن را در آرایه قرار دهید.

### افزودن مقدار به یک session آرایه ای

کد پی ایچ پی:

```
Session::push('user.teams', 'developers');
```

بازیابی مقدار session با متد get امکانپذیر است.

کد پی ایچ پی:

```
$value = Session::get('key');
$value = session('key');
```

در صورتی که session مقداری نداشت می توانیم برای آن یک مقدار پیش فرض تعریف کنیم

کد پی ایچ پی:

```
$value = Session::get('key', 'default');
$value = Session::get('key', function() { return 'default'; });
```

گرفتن مقدار یک session و بلافاصله حذف آن با متد pull امکانپذیر است:

کد پی ایچ پی:

```
$value = Session::pull('key', 'default');
```

با متد `all` می توانیم به تمام مقادیر سشن ها را در یک آرایه بازیابی کنیم.

کد پی اچ پی:

```
$data = Session::all();
```

برای حذف یک `session` خاص از متد `forget` که نام سشن را به آن می دهیم استفاده می کنیم. برای حذف تمامی `session` ها از `flush` استفاده میکنیم.

کد پی اچ پی:

```
Session::forget('key');
```

```
Session::flush();
```

برای امنیت بیشتر سشن ها می توانید از متد `regenerate` برای تولید دوباره `session id` استفاده کنید:

کد پی اچ پی:

```
Session::regenerate();
```

## داده های فلش

سشن ها بعد از تولید تا وقتی که مرورگر بسته نشود از بین نمی روند. در لاراول `session`هایی به نام فلش وجود دارند که فقط برای یک درخواست معتبر هستند و بلافاصله در درخواست بعدی از بین میروند که مناسب برای ایجاد پیغام های خطا می باشند. مانند مثال زیر آنها را تولید می کنیم و به مانند سشن های دیگر بازیابی میکنیم.

کد پی اچ پی:

```
Session::flash('key', 'value');
```

## ذخیره سشن ها در دیتابیس

`session`ها به طور پیش فرض در فایل ذخیره می شوند. شما می توانید آنها در چند جای مختلف از جمله دیتابیس ذخیره کنید که هرکدام در کاربردهای خاصی استفاده می شوند. در صورتی که میخواهید سشن ها را در دیتابیس ذخیره کنید کافی است این سه دستور را به ترتیب در ترمنال تایپ و اجرا کنید:

```
php artisan session:table
composer dump-autoload
php artisan migrate
```

سپس در فایل env مقدار SESSION\_DRIVER را به database تغییر دهید .

## اعتبار سنجی در لاراول ۵

توی این پست یک مثال کاربردی از اعتبار سنجی فرم ها در فریم ورک لاراول رو خواهیم داشت. برای این منظور ابتدا یک فرم رو در فایل view مثلا به نام form.blade.php در پوشه resources/views ایجاد می کنم و کدهای فرم را به این صورت می نویسم:

کد پی ایچ پی:

```
<ul>
  @foreach($errors->all('<li>:message</li>') as $error)
    {!! $error !!}
  @endforeach
</ul>
<form action="{ url('test') }" method="post">
  <input type="hidden" name="_token" value="{ csrf_token() }">
  <label for="name">Name</label>
  <input type="text" name="name" id="name" value="{ old('name') }">

  <label for="email">Email</label>
  <input type="text" name="email" id="email" value="{ old('email') }">

  <label for="age">Age</label>
  <input type="text" name="age" id="age" value="{ old('age') }">

  <input type="submit" value="Submit">
</form>
```

همینطور که مشاهده میکنید اکشن فرم را به مسیر test تعیین کردم. برای فرم هایتان باید حتما یک توکن تعیین کنید که یک فیلد مخفی با نام token\_ است و مقدار آن توسط تابع csrf\_token ایجاد می شود و برای جلوگیری از حملات csrf به کار می رود. برای هر تکست باکس هم مقدار آن را با تابع کمکی old مقداردهی کردم تا در صورت ریدایرکت بک شدن درخواست مقادیر قبلی فرم حفظ شوند. خب حالا باید توی فایل routes.php دوتا مسیر تعریف کنیم. مسیر get که فایل فرم را رندر میکند و در مرورگر نمایش می دهد و post هم که مقادیر بعد از سابمیت به آن ارسال می شوند.

## کد پی اچ پی:

```
Route::get('test', function(){
    return view('form');
});

Route::post('test', function(){
});
```

من برای طولانی نشدن مثال در همین فایل routes اعتبارسنجی رو انجام میدم اما شما بهتره برای رعایت اصول MVC این اعمال را داخل کنترلرها انجام بدین.

حالا اعتبارسنجی رو به این صورت انجام میدم:

## کد پی اچ پی:

```
Route::post('test', function(){
    $validator = Validator::make(
        Request::all(),
        [
            'name' => 'required',
            'email' => 'required|email|unique:users',
            'age' => 'numeric',
        ]
    );

    if($validator->fails()){
        return redirect()->back()->withErrors($validator->errors())->withInput();
    }
});
```

همانطور که می بینید از کلاس Validator و متد make استفاده کردم. این متد دوتا پارامتر آرایه ای می گیرد که اولی آرایه ای از مقادیر هست که از فرم ارسال کرده ایم و دومی هم آرایه ای هست که قوانین اعتبارسنجی را برای هر فیلد تعریف می کنیم. چیزی که اینجا جدیده نحوه نوشتن قوانین اعتبارسنجی هست که یک آرایه هست که باید کلید آن نام اون فیلد فرم و مقدار اون قوانین اون فیلد باشد و هر قانون را هم با کاراکتر | از هم جدا میکنیم required. یعنی الزامی بودن فرم و email یعنی یک آدرس ایمیل معتبر باشد یا numeric یعنی مقدار باید عددی باشد و ... در اینجا از یک قانون به نام unique برای فیلد email قرار دادیم که در جدول users بررسی می کند که مقدار ایمیل وارد شده در جدول قبلا ثبت نشده باشد. البته باید نام ستون ایمیل در جدول با نام فیلد یکی باشد و گرنه باید نام ستون را هم جلوی قانون اضافه کنیم. در نهایت با متد fails بررسی میکنیم اگر اعتبارسنجی دارای خطا بود به صفحه قبل ریدایرکت شود. پیغام های خطا و مقادیر قبلی فرم هم ارسال شوند.

حالا یک روش خیلی ساده تر از قبلی رو بهتون میگم که به جای استفاده از کلاس Validator داخل کنترلر از متد validate خود کنترلر استفاده کنید:

## کد پی اچ پی:

```
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);
    //
}
```

تو این روش اعتبارسنجی انجام می شود و اگر خطای اعتبارسنجی نداشت که به ادامه کار می پردازد و گرنه خودش اتوماتیک به صفحه قبلی ریدایرکت میکند و پیغام های خطا را هم به آنجا ارسال می کند. همه برنامه نویسان حرفه ای بدنبال این هستند که همیشه حداقل کد رو بنویسن پس اگر توی کلاس کنترلر مورنظر چندین بار از اعتبارسنجی در اکشن های مختلف می خواهید استفاده کنید باز روش بهتری هست که قوانین رو در یک کلاس request ایجاد کنید. ابتدا با دستور زیر در ترمینال یک کلاس request با نام دلخواه ایجاد کنید:

## کد:

```
php artisan make:request StoreBlogPostRequest
```

توجه داشته باشید این کلاس حتما باید از کلاس Request ارث برده شود. حالا توی متد rules اون کلاس قوانین رو تعریف کنیم:

## کد پی اچ پی:

```
public function rules()
{
    return [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ];
}
```

کافیست تو هر اکشن کنترلی که می خواهیم اعتبارسنجی انجام شود از این کلاس استفاده کنیم.

## کد پی اچ پی:

```
public function store(StoreBlogPostRequest $request)
{
    // The incoming request is valid...
}
```

درخواست ها ابتدا اعتبارسنجی می شوند در صورتی که بدون خطا باشند وارد اکشن می شوند و گرنه به طور اتوماتیک به صفحه قبلی ریدایرکت و پیغام های خطا هم قابل دسترسی هستند.

## نمایش پیام های خطا در view

کد پی ایچ پی:

```
echo $errors->first('email');

foreach ($errors->all() as $error)
{
    //
}
```

در صورتی که فقط خطای فیلد خاصی را بخواهیم نمایش دهیم مانند مثال اول و اگر همه پیام ها را نمایش دهیم به مانند مثال دوم عمل میکنیم. همچنین می توانیم پیام های خطا را در قالب یک تگ HTML نمایش دهیم که من در مثالم به این صورت عمل کردم:

کد پی ایچ پی:

```
<ul>
    @foreach($errors->all('<li class="error">:message</li>') as $error)
        {!! $error !!}
    @endforeach
</ul>
```

## ایجاد یک قانون اعتبار سنجی دلخواه

اگر قانون مورد نظر شما در قوانین موجود لاااول وجود نداشت می توانید با استفاده از متد extend این قانون را ایجاد کنید:

کد پی ایچ پی:

```
Validator::extend('alpha_spaces', function($attribute, $value)
{
    return preg_match('/^[^pL\s]+$u', $value);
});
```

مثلا قانونی که من نیاز داشتم مجاز بودن حروف الفبا و فاصله در یک مقدار بود که در بالا تعریف کردم.

## ایجاد پیام خطای دلخواه برای قوانین اعتبارسنجی

پیغام ها خطا به طور پیش فرض در مسیر resources/lang/en و فایل validation.php تعریف شده اند و به زبان انگلیسی هستند. ما می توانیم یک آرایه تعریف کنیم که کلید آن نام قانون و مقدار آن پیغام خطای مورد نظر شما می باشد و این آرایه را به عنوان پارامتر سوم به متد make بدهیم.

## کد پی ایچ پی:

```
$messages = [
    'same' => 'The :attribute and :other must match.',
    'size' => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in' => 'The :attribute must be one of the following types: :values',
];

$validator = Validator::make($input, $rules, $messages);
```

البته راه بهتری پیشنهاد میکنم به جای اینکه در هر اکشن بخواهید این پیغام هارا ست کنید بهتر است داخل مسیر resources/lang پوشه به نام fa ایجاد کنیم و همه محتویات پوشه en را داخل آن کپی کنیم و سپس داخل فایل validation.php پیغام های خطای هر قانون را به فارسی و دلخواه خودتان ست کنید. همچنین داخل آرایه attributes داخل همان فایل هم نام فیلدهای فرم که به طور پیش فرض از خاصیت name هر تکست باکس گرفته می شود را به دلخواه خودتان تغییر دهید.

به مثال زیر توجه کنید:

## کد پی ایچ پی:

```
"required" => "پر کردن آن الزامی است :attribute فیلد"

    'attributes' => [
        'name' => 'نام',
        'email' => 'آدرس ایمیل',
        'age' => 'سن',
    ],
```

برای قانون required یک پیغام دلخواه و نام دلخواهی برای فیلدها در نظر گرفتیم. برای استفاده از این پیغام های دلخواه چون من این پوشه را fa نامگذاری کردم باید داخل فایل app.php در پوشه config آیتم locale را به fa تغییر دهید.



## کار با دیتابیس در لارا اول ۵

### مباحث پایه کار با دیتابیس

یکی از مزیت های فریم ورک لارا اول کار با دیتابیس آن است که بسیار ساده است و متدهای زیادی برای عملیات های مختلف دارد. برای اعمال تنظیمات دیتابیس خود باید داخل فایل env. و همچنین در پوشه config و فایل database.php تنظیمات مورد نظر خود را اعمال کنید. به طور پیش فرض لارا اول از mysql استفاده می کند اما از دیتابیس های MySQL , Postgres, SQLite و SQL Server هم پشتیبانی می کند و می توانیم از هر یک از آنها استفاده کنیم.

### اجرای کوئری با کلاس DB

در لارا اول به سادگی می توانیم با استفاده از کلاس DB و نوشتن کوئری به صورت prepared statements عمل مورد نظرمان را انجام دهیم.

با استفاده از متد select می توانیم رکوردهای داخل یک جدول را بازیابی کنیم و خروجی آن یک آرایه است. پارامتر دوم متد select هم یک آرایه از مقادیر است که در صورتی که کوئری نیاز به bind کردن مقداری داشته باشد از آن استفاده میکنیم. نحوه استفاده از آن را به دوشکل مختلف می بینید:

کد پی ایچ پی:

```
$results = DB::select('select * from users where id = ?', [1]);
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

insert , update , delete

برای درج در جدول از متد insert و برای به روز رسانی از update و حذف از جدول delete را استفاده میکنیم:

کد پی ایچ پی:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
DB::update('update users set votes = 100 where name = ?', ['John']);
DB::delete('delete from users where id =:id', ['id' => 1]);
```

نکته: متدهای update و delete تعداد رکوردهایی که با این کوئری تغییر یافتند یا حذف شدند را برمیگرداند.

اگر کوئری غیر از ۴ عمل اصلی دیتابیس بود می توانیم از متد ststatement استفاده کنیم:

کد پی اچ پی:

```
DB::statement('drop table users');
```

برای تراکنش هم می توانید از متد transaction استفاده کنید و عملیات مورنظرتان را داخل تابع که به آن می دهیم را انجام دهیم. در صورتی که هر یک از کوئری ها با خطایی مواجه شوند و اجرا نشوند به طور اتوماتیک تمام کوئری های اجرا شده به عقب بر میگردند که مناسب برای عملیات های مالی می باشد.

کد پی اچ پی:

```
DB::transaction(function()
{
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

در صورتی که در برنامه تان از چند اتصال به دیتابیس استفاده می کنید با استفاده از متد connection و دادن نام اتصال به آن به عنوان پارامتر از آن استفاده کنیم:

کد پی اچ پی:

```
$users = DB::connection('foo')->select(...);
```

## کار با دیتابیس با Query Builder

روش بهتر و آسانتر برای کار با دیتابیس در لاراول به جای نوشتن کامل کوئری استفاده از Query Builder است. شما می توانید اکثر عملیات های دیتابیس را در برنامه تان انجام بدهید و این کوئری ها در همه دیتابیس هایی که لاراول ساپورت می کند کار کند. در ضمن کوئری بیلدر لاراول از bind کردن پارامترها استفاده می کند که برنامه تان را در برابر حملات SQL Injection محافظت میکند.

### SELECT

برای انتخاب تمامی رکوردهای یک جدول ابتدا نام جدول موردنظر را به متد table و سپس با متد get رکوردها را واکنشی میکنیم.

کد پی اچ پی:

```
$users = DB::table('users')->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

برای استفاده از شرط در کوئری از متد where استفاده می کنیم و این متد سه پارامتر میگیرد که اولی نام

ستون موردنظر و دومی operator شرط (= , > , < , <= , ... ) و سومین پارامتر هم مقدار موردنظر است. در صورتی که پارامتر دوم را ننویسیم به صورت پیش فرض عملگر = در نظر گرفته می شود. متد first هم اولین رکورد که با شرط فوق همخوانی داشته باشد را برمیگرداند که برای بازیابی یک رکورد استفاده می شود. در صورتی که چند رکورد را بخواهیم بازیابی کنیم از متد get استفاده میکنیم.

کد پی اچ پی:

```
$user = DB::table('users')->where('name', 'John')->first();
var_dump($user->name);$users = DB::table('users')->where('votes', '>', 100)->get();
```

در صورتی که بخواهیم مقدار یک ستون خاص را که در یک شرط صدق میکند را بازیابی کنیم از متد pluck استفاده و نام ستون را به آن میدهیم. اگر بخواهیم لیست مقادیر یک ستون را واکنشی کنیم از متد lists استفاده و مقدار ستون را به عنوان پارامتر به آن میدهیم که خروجی آن یک آرایه است و می توانیم نام ستون دیگری را هم به عنوان پارامتر دوم به آن بدهیم تا کلید آرایه مقادیر آن ستون باشند.

کد پی اچ پی:

```
$name = DB::table('users')->where('name', 'John')->pluck('name');
$roles = DB::table('roles')->lists('title');
$roles = DB::table('roles')->lists('title', 'name');
```

## استفاده از OR یا AND برای جدا کردن شرط ها

برای این کار کافی است بعد از متد where که نوشتیم متد orWhere را استفاده کنیم:

کد پی اچ پی:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
    عبارت بالا معادل کوئری زیر است:
    SELECT * FROM users WHERE votes > 100 OR name = 'john'
```

اگر دوباره از متد where استفاده کنیم معادل AND در نظر گرفته می شود. متدهای بسیار زیادی وجود دارند که به علت طولانی شدن میجست و وجود مثال ها به طور واضح در داکيومنت برای اطلاعات بیشتر به اینجا مراجعه کنید.

## استفاده از متدهای جادویی شرط

روش بهتر و با کدنویسی کمتر استفاده از متدهای جادویی هست. در مثال های زیر کوئری های معادل آنها را هم نوشته ام:

کد پی اچ پی:

```
//SELECT * FROM users WHERE id=1 LIMIT 1;
$admin = DB::table('users')->whereId(1)->first();

//SELECT * FROM users WHERE id=2 AND email = 'john@doe.com' LIMIT 1;
$joh = DB::table('users')
    ->whereIdAndEmail(2, 'john@doe.com')
    ->first(); //

//SELECT * FROM users WHERE name='Jane' OR age = 22 LIMIT 1;
$jane = DB::table('users')
    ->whereNameOrAge('Jane', 22)
    ->first();
```

## استفاده از Order By و Group By و Having با کوئری بیلدر

کد پی اچ پی:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->groupBy('count')
    ->having('count', '>', 100)
    ->get();
```

همچنین می توانیم از LIMIT به همراه آفست در کوئری استفاده کنیم .

کد پی اچ پی:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

در مثال بالا کوئری میگوید که از رکورد دهم در جدول users را انتخاب کن و تا ۵ رکورد را واکنشی کن. (شماره گذاری رکوردها از صفر شروع میشود)

## JOIN کردن

با متد join می توانید دو یا چند جدول را باهم JOIN کنید. این متد ۴ پارامتر می گیرد که اولی جدولی که میخواهیم به آن پیوند بزنیم و پارامترهای بعدی فیلدهایی که باید باهم مساوی باشند را قرار میدهم.

کد پی اچ پی:

```
DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.id', 'contacts.phone', 'orders.price')
    ->get();
```

در مثال بالا به سه جدول orders, users, contacts پیوند زده شده است.

با کوئری بیلدر می توانیم با توابع جمعی (count, max, min, ...) تمام مقادیر اسکالر یک ستون را محاسبه کرده و مقداری اسکالر تولید می کند

کد پی اچ پی:

```
$users = DB::table('users')->count();
$price = DB::table('orders')->max('price');
$price = DB::table('orders')->min('price');
$price = DB::table('orders')->avg('price');
$total = DB::table('users')->sum('votes');
```

## درج کردن (INSERT)

با استفاده از متد insert می توانیم در جدول مورد نظر مقادیری را درج کنیم. مقادیر را در آرایه قرار می دهیم و به عنوان پارامتر به آن می دهیم. کلیدهای آرایه نام ستون جدول مورد نظر است.

کد پی اچ پی:

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);

$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);

DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

اگر در جدولتان فیلد id به صورت Auto-increment است می توانید از متد insertGetId استفاده کنید که بعد از درج کوئری id که تولید شده را به عنوان خروجی برمیگرداند. در مثال سوم در بالا هم همانطور که می بینید در صورتی که بخواهید چندین رکورد را باهم درج کنید کفایت رکوردها را به عنوان پارامتر به متد insert بدهیم و با ویرگول ازهم جدا کنیم.

## به روزرسانی (UPDATE)

با استفاده از متد update که یک آرایه به آن می دهیم که کلیدهای آن نام ستون مورد نظر در جدول و مقادیر آن هم مقدار جدید می باشد رکوردها را آپدیت کنیم.

## کد پی ایچ پی:

```
DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);
```

همچنین می توانیم با متد `increment` مقدار ستونی را یک واحد افزایش دهیم یا با ذکر یک پارامتر دوم تعداد افزایش را به طور مثال در مثال زیر ۵ واحد مشخص کنیم. متد `decrement` هم مقدار را کاهش می دهد و مانند متد قبلی عمل میکند.

## کد پی ایچ پی:

```
DB::table('users')->increment('votes');
DB::table('users')->increment('votes', 5);
DB::table('users')->decrement('votes');
DB::table('users')->decrement('votes', 5);
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

در مثال آخر هم همانطور که می بینید هم می توانیم عمل افزایش را انجام دهیم و هم آپدیت سایر مقادیر ستون های جدول را که به عنوان پارامتر سوم و از نوع آرایه به آن می دهیم.

## حذف کردن (Delete)

در لاراوِل با استفاده از متد `delete` می توانیم رکوردی یا همه رکوردهای جدول را حذف کنیم. اگر از شرط استفاده نکنیم همه رکوردهای جدول حذف می شوند. با استفاده از متد `truncate` هم می توانیم همه مقادیر یک جدول را حذف کنیم با این تفاوت که `truncate` هیچ شرطی نمیگیره و سریعتر از `delete` هست یا تفاوت دیگر آن این است که `id` های اختصاص داده شده به رکوردها را هم `reset` میکند ولی در `delete` اینگونه نیست.

## کد پی ایچ پی:

```
DB::table('users')->where('votes', '<', 100)->delete();
DB::table('users')->delete();
DB::table('users')->truncate();
```

با استفاده از متد `union` می توانیم دو کوئری را باهم اجتماع کنیم:

## کد پی ایچ پی:

```
$first = DB::table('users')->whereNull('first_name');
$users = DB::table('users')->whereNull('last_name')->union($first)->get();
```

## قفل کردن جدول هنگام اجرای عملیات

در صورتی که قصد دارید در هنگام انجام عملیات SELECT جدول قفل شود میتوانیم به صورت زیر عمل کنیم:

کد پی ایچ پی:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

با استفاده از متد sharedLock جدول را به طور کامل قفل می کنیم و با متد lockForUpdate جدول را هنگام عملیات SELECT فقط برای به روزرسانی قفل می کنیم.

## Eloquent در لاراوول ۵

در لاراوول می توانیم با استفاده از Eloquent که پیاده سازی شده از الگوی طراحی ActiveRecord است خیلی ساده تر با دیتابیس کار کنیم. در این روش هر جدول در دیتابیس با یک کلاس Model در ارتباط است.

برای شروع کار با Eloquent باید ابتدا یک کلاس مدل از جدول ایجاد کنیم. کلاس های مدل را داخل پوشه app قرار میدهیم. با تایپ این دستور در ترمینال می توانیم یک مدل ایجاد کنیم:

کد:

```
php artisan make:model User
```

نام مدل را همیشه به صورت PascalCase بنویسید. به طور پیش فرض مدل با جدولی که مشابه نام مدل است. اما فقط یک S به آخر اضافه شده متناظر است. مثلا مدل User با جدول users در دیتابیس مرتبط است. کلاس ایجاد شده باید محتوای آن به شکل زیر باشد:

کد پی ایچ پی:

```
<?php namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model {}
```

در صورتی که نام جدولی که با کلاس مدل از قانونی که در بالا گفتم تبعیت نمیکند میتوانید در کلاس با استفاده از پراپرتی table نام مورد نظر را ست کنیم.

کد پی ایچ پی:

```
class User extends Model {
    protected $table = 'my_users';
    public $timestamps = false;
}
```

همچنین در جداول باید دو ستون تاریخ و زمان به نام های `created_at` , `updated_at` وجود داشته باشند که هنگام ایجاد رکورد یا به روزرسانی آن مقداردهی می شوند اما شما می توانید با `false` قراردادن پراپرتی `timestamps` از ایجاد این ستون ها در جدول صرف نظر کنید.

حالا می توانیم به راحتی از کلاس مدل برای عملیات های دیتابیس استفاده کنیم. در `Eloquent` می توانیم از همه متدهای `Query Builder` استفاده کنیم. برای بازیابی کلیه رکوردها از متد `all` استفاده میکنیم.

کد پی ای پی:

```
$users = User::all();
```

رکوردی را با داشتن `id` آن می توانیم با متد `find` بازیابی کنیم:

کد پی ای پی:

```
$user = User::find(1);
```

```
var_dump($user->name);
```

در صورتی که هنگام بازیابی رکوردها مقداری یافت شد و خواستیم یک خطای استثناء تولید شود کلمه `OrFail` را به انتهای متد مورد نظر اضافه میکنیم:

کد پی ای پی:

```
$model = User::findOrFail(1);
```

```
$model = User::where('votes', '>', 100)->firstOrFail();
```

از تمام متدهایی که در `Query Builder` یاد گرفتیم میتوانیم در `Eloquent` هم استفاده کنیم:

کد پی ای پی:

```
$users = User::where('votes', '>', 100)->take(10)->get();
foreach ($users as $user)
{
    var_dump($user->name);
}
```

## درج کردن با Eloquent

ابتدا یک شی از کلاس مدل ایجاد میکنیم و سپس با استفاده از شی ایجاد شده `attribute` های مدل که همان نام ستونهای جدول هستند را با مقدار جدید مقداردهی میکنیم و سپس با صدازدن متد `save` رکورد جدید را ایجاد میکنیم.

کد پی ای پی:

```
$user = new User;
$user->name = 'John';
$user->save();
$insertedId = $user->id;
```



بعد از درج هم میتوانیم به id اختصاص داده شده به این رکورد دسترسی داشته باشیم. همچنین می توانیم با استفاده از متد create یک رکورد جدید را به جدول اضافه کنیم که به این روش mass-assignment گفته میشود که البته بایستی در کلاس مدل یک پراپرتی protected به نام guarded ایجاد کنیم که یک لیست سیاه می باشد و اجازه تغییر فیلدهای موردنظر را به کاربر نمیدهد. پراپرتی fillable برعکس guarded است و یک لیست سفید برای عملیات mass-assignment ایجاد میکند.

کد پی اچ پی:

```
// Create a new user in the database...
$user = User::create(['name' => 'John']);

// Retrieve the user by the attributes, or create it if it doesn't exist...
$user = User::firstOrCreate(['name' => 'John']);

// Retrieve the user by the attributes, or instantiate a new instance...
$user = User::firstOrCreate(['name' => 'John']);
```

همچنین می توانیم از متدهای جادویی هم استفاده کنیم که مثلا در مثال دوم رکوردی را با این مقدار بازیابی کند و اگر وجود نداشت آن را ایجاد کند.

## به روزرسانی رکوردها

برای آپدیت هم مشابه درج کردن عمل میکنیم فقط با این تفاوت که به جای ایجاد شی از کلاس مدل باید رکورد مورد نظر را ابتدا بازیابی کنید. مثلا در مثال زیر رکورد با id برابر ۱ را بازیابی کرده و سپس فیلد email را با مقدار جدیدی آپدیت میکند:

کد پی اچ پی:

```
$user = User::find(1);
$user->email = 'john@foo.com';
$user->save();
```

## حذف رکوردها

با استفاده از متد delete می توانید رکورد بازیابی شده را به راحتی حذف کنید.

کد پی اچ پی:

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

همچنین روش آسانتر استفاده از متد destroy است که id رکورد را به ان میدهیم. در صورتی که تعداد id ها بیش از یکی بود هم می توانید در آرایه قرار دهید و به عنوان پارامتر به متد دهید و یا اینکه هرکدام را با ویرگول از هم جدا کنید.

```
User::destroy(1);
User::destroy([1, 2, 3]);
User::destroy(1, 2, 3);
```

## Relationships جداول در لاراوِل ۵

### ارتباطات (Relationships)

در فریم ورک لاراوِل تقریباً همه جداول موجود در دیتابیس بایکدیگر ارتباط دارند. ارتباطات می تواند از انواع یک به یک و یک به چند و چند به چند باشد. در لاراوِل با Eloquent به راحتی می توانید این ارتباط ها را مدیریت و با آنها کار کنید. در این پست دو نمونه رایج ارتباط یک به چند (One-to-Many) و چند به چند (Many-to-Many) که اکثر ارتباطها در جداول دیتابیس به این صورت است را مثال خواهیم زد.

### ارتباط One To Many

برای مثال یک وبلاگ را در نظر بگیرید که دارای یک جدول به نام posts و یک جدول هم به نام comments هست. هر پست در بلاگ می تواند دارای چند کامنت و هر کامنت هم فقط به یک پست تعلق دارد پس این ارتباط یک به چند است .  
داخل کلاس مدل (Post که سمت یک ارتباط است) ابتدا ارتباط به مدل Comment را با افزودن یک متد همنام با جدول متناظرش مثلاً به نام comments به این صورت پیاده سازی میکنیم:

کد پی اچ پی:

```
class Post extends Model {

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

}
```

با استفاده از متد hasMany کلاسی که با آن ارتباط چندی دارد را به عنوان پارامتر به آن میدهم. همچنین باید داخل کلاس مدل Comment هم متدی همنام کلاس متناظرش مثلاً post ایجاد کرده و سپس با استفاده از متد belongsTo کلاس Post را به عنوان پارامتر به آن میدهم.

کد پی اچ پی:

```
class Comment extends Model {

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

اکنون همانند مثال زیر می توانید تمام کامنت های پستی با id برابر ۱ را بازیابی کنید. همچنین می توانید از سایر متدها همچون شرط هم استفاده کنید.

کد پی اچ پی:

```
$comments = Post::find(1)->comments;
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

نکته: نام کلید خارجی باید به صورتی باشد که ابتدا نام جدولی که از آن ارجاع می شود بدون s و سپس کلمه id\_ به انتهای آن افزوده شود مثلا برای مثال بالا کلید خارجی باید post\_id باشد وگرنه باید در متد hasMany کلید خارجی را هم مشخص کنیم:

کد پی اچ پی:

```
return $this->hasMany('App\Comment', 'foreign_key');
```

## ارتباط Many To Many

برای پیاده سازی این نوع ارتباط فرض کنید یک جدول به نام users داریم و یک جدول هم به نام roles. هر کاربر می تواند چندین نقش داشته باشد و هر نقش هم میتواند به چندین کاربر تعلق داشته باشد. پس باید یک جدول واسط هم برای این دو جدول به نام role\_user داشته باشیم. دقت کنید نام این جدول باید ترکیبی از نام دو جدول قبلی اما بدون s آخر آنها باشد که با \_ از هم جدا شده اند. سپس کلید های خارجی user\_id و role\_id هم در این جدول ایجاد می شوند.

در مدل User یک متد همانم جدولی که با آن ارتباط دارد ایجاد میکنیم و سپس با استفاده از متد belongsToMany کلاس مدل Role را به آن میدهم.

کد پی اچ پی:

```
class User extends Model {
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

در کلاس مدل Role هم مانند بالا عمل میکنیم:

کد پی اچ پی:

```
class Role extends Model {
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

حالا به راحتی می توانیم تمامی نقش های یک کاربر را بازیابی کنیم:

کد پی اچ پی:

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

## درج کردن در جدول رابطه دار

فرض کنید می خواهیم یک کامنت را در جدول comments درج کنیم. همانطور که قبلا مثال زدیم جدول posts با جدول comments دارای ارتباط یک به چند است و ستون post\_id در جدول comment کلید خارجی است. همانند مثال زیر می توانید به روش mass-assignment رکوردی را در ج کنید به طوری که در فیلد post\_id به طور اتوماتیک با توجه به پست مورد نظر id آن ثبت خواهد شد.

کد پی اچ پی:

```
$comment = new Comment(['message' => 'A new comment.']);
$post = Post::find(1);
$comment = $post->comments()->save($comment);
```

نکته: در این روش درج باید حتما پراپرتی guarded\$ را هم در کلاس مدل مورد نظر که میخواهید عمل درج را انجام دهید ست کنید تا ستون هایی که قرار نیست توسط کاربر درج شود محافظت شوند. به طور مثال در زیر من اینگونه آن را تعریف کردم:

کد پی اچ پی:

```
public $guarded = ['id' , 'post_id'];
```

همچنین می توانید تعداد زیادی کامنت را هم به روش بالا درج کنید. هررکورد را داخل یک آرایه قرار می دهیم و همچنین به جای متد save از saveMany استفاده میکنیم.

کد پی اچ پی:

```
$comments = [
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another comment.']),
    new Comment(['message' => 'The latest comment.'])
];
$post = Post::find(1);
$post->comments()->saveMany($comments);
```

بعضی مواقع نیاز داریم که هنگام select کردن رکوردها خروجی را در قالب آرایه یا JSON داشته باشیم که Eloquent دارای متدهایی برای این کار می باشد.

با استفاده از متد toArray می توانیم خروجی هر کوئری را به یک آرایه تبدیل کنیم

کد پی اچ پی:

```
$user = User::with('roles')->first();
return $user->toArray();
```

با متد toJson هم خروجی را به JSON تبدیل می کنیم:

کد پی اچ پی:

```
return User::find(1)->toJson();
```

## صفحه بندی در لارا اول ۵

هنگامی که تعداد رکوردهایی که می خواهید در یک صفحه وب نمایش دهید زیاد می باشد بهترین روش برای مدیریت تعداد نمایش در هر صفحه صفحه بندی کردن است. در لارا اول شما آسان تر از سایر فریمورک ها می توانید این کار را انجام دهید. کد HTML ای هم که برای نمایش صفحه بندی تولید می شود سازگار با Bootstrap Twitter می باشد.

هنگام بازیابی رکوردها از دیتابیس کافی است از متد paginate استفاده کنیم و تعداد آیتم های قابل نمایش در هر صفحه را هم به عنوان پارامتر به آن بدهیم:

کد پی اچ پی:

```
$users = DB::table('users')->paginate(15);
$allUsers = User::paginate(15);

$someUsers = User::where('votes', '>', 100)->paginate(15);
```

در مثال های فوق هم با روش کوئری بیلدر و هم Eloquent اینکار را انجام داده ایم و تعداد آیتم ها را ۱۵ تعیین کردیم.

حالا فرض کنید تمام کاربران را از دیتابیس واکنشی کردیم و به صفحه view با متغیری به نام users ارسال کردیم. داخل ویو موردنظر کدهای زیر را قرار می دهیم:

کد پی ایچ پی:

```
<div class="container">
  <?php foreach ($users as $user): ?>
    <?php echo $user->name; ?>
  <?php endforeach; ?>
</div>

<?php echo $users->render(); ?>
```

با استفاده از حلقه foreach نام کاربران را نمایش می دهیم. برای نمایش کد HTML مربوط به صفحه بندی هم از متد render استفاده می کنیم و آن را چاپ می کنیم. البته مثال بالا به روش php نوشته شده و شما بهتر است از موتور قالب Balde استفاده کنید. با CSS می توانید قالب نمایش صفحه بندی را به دلخواه خودتان تغییر دهید. غیر از متد render متدهای دیگر هم وجود دارند که می توانید اطلاعات بیشتری را بدست آورید به طور مثال currentPage شماره صفحه جاری را نمایش می دهد و lastPage شماره آخرین صفحه و ... اگر فقط می خواهید لینک Next و Previous نمایش داده شود و صفحه بندی ساده ای باشد از متد simplePaginate هنگام واکنشی استفاده کنید .

کد پی ایچ پی:

```
$someUsers = User::where('votes', '>', 100)->simplePaginate(15);
```

به طور پیش فرض URL هنگام صفحه بندی مثلا به صورت ?page=2 خواهد بود شما می توانید با متد setPath یک URL دلخواه هنگام نمایش صفحه بندی ایجاد کنید:

کد پی ایچ پی:

```
$users = User::paginate();
$users->setPath('custom/url');
```

در مثال بالا آدرس URL به صورت http://example.com/custom/url?page=2 نمایش داده می شود.

به انتهای URL می توانیم کوئری استرینگ هم اضافه کنیم. هنگام نمایش صفحه بندی با استفاده از متد append که داده های کوئری استرینگ را به عنوان آرایه به آن می دهیم:

کد پی ایچ پی:

```
<?php echo $users->appends(['sort' => 'votes'])->render(); ?>
```

در مثال بالا آدرس URL به صورت http://example.com/something?page=2&sort=votes نمایش داده میشود.

همچنین می توانیم با متد `fragment` یک آدرس `fragment` را به انتهای URL اضافه کنیم.

کد پی اچ پی:

```
<?php echo $users->fragment('foo')->render(); ?>
```

در مثال بالا URL به صورت `http://example.com/something?page=2#foo` نمایش داده می شود.

## Migration در لاراوول ۵

migration یک نوع کنترل ورژن برای دیتابیس برنامه تان است. به تیم برنامه نویسی شما این اجازه رو میدهد که شما (Schema) دیتابیس رو طراحی کنند و تغییر بدهند. با این ابزار شما می توانید نسخه های مختلفی از دیتابیس رو هم داشته باشید که با یکسری دستورات میتونید به نسخه های مختلف سوئیچ کنید. یکی از مزیت های دیگر آن مهم نبودن نوع دیتابیس است و دستورات شما با همه دیتابیس هایی که لاراوول ساپورت میکند کار خواهد کرد migration. به طور معمول با استفاده از Schema Builder کار میکند.

### ایجاد یک migration

در فریم ورک لاراوول با استفاده از دستور `make:migration` و تایپ آن در ترمینال می توانیم یک migration جدید ایجاد کنیم:

کد:

```
php artisan make:migration create_users_table
```

migration ایجاد شده در مسیر پوشه `database/migrations` قرار می گیرد. اکنون با استفاده از دستورات Schema Builder می توانیم schema جدول موردنظرمان را طراحی کنیم.

```

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

class CreateUsersTable extends Migration {
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function(Blueprint $table)
        {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password', 60);
            $table->timestamps();
        });
    }

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

```

همانطور که می بینید کلاس migration دارای دو متد به نام up و down می باشد. از up برای ایجاد جدول و افزودن یا تغییر ستون ها و از down برای عمل برعکس آن استفاده میکنیم مثلا اگر جدولی را ایجاد کرده ایم در متد down آن را حذف میکنیم. از کلاس Schema و متد create برای ایجاد جدول استفاده میکنیم که دو پارامتر می گیرد اولی نام جدول می باشد و دومی یک تابع بی نام می باشد که آبجکت table\$ را به عنوان پارامتر میگیرد. داخل تابع ستون ها را تعریف میکنیم. به صورتی که متدهای آبجکت table\$ نوع ستون و پارامتر ورودی آنها نام ستون را مشخص میکند.

مثلا متد increments نوع int و کلید اصلی به همراه AutoIncrement در نظر میگیرد.

نوع string در نظر گرفته می شود و پارامتر دومی هم میشود به آن داد که طول رشته را مشخص میکند.

unique ستون را به ایندکس unique تبدیل میکند.

timestamps فیلدهای زمانی created\_at و updated\_at را ایجاد میکند.

در تابع down هم با استفاده از متد drop مشخص کردیم جدول users حذف شود.

با استفاده از دستور زیر می توانید migration ایجاد شده را اجرا کنید:

کد:

```
php artisan migrate
```



با اجرای دستور بالا جدول موردنظر در دیتابیس ایجاد خواهد شد. البته دستور بالا تمام migration هایی که اجرا نشده اند را اجرا میکند.

و با استفاده از دستور زیر می توانید دستورات `down` را اجرا کنید و با توجه به مثال بالا جدول `users` حذف می شود:

کد:

```
php artisan migrate:rollback
```

نکته: اگر در هنگام اجرای دستور `migrate` پیغام خطای `"class not found"` داد از دستور زیر قبل از دستور `migrate` در ترمینال استفاده کنید:

کد:

```
composer dump-autoload
```

در هنگام ایجاد migration می توانید نام جدول را هم در دستور مشخص کنید.

کد:

```
php artisan make:migration create_users_table --create=users
```

برای نامگذاری فایل migration معمولا از یک نام با مسما که نشانه عملیات موردنظرمان است استفاده میکنیم مثلا برای افزودن یک ستون جدید به نام `votes` در جدول `users` به این صورت فایل را نامگذاری و ایجاد میکنیم:

کد:

```
php artisan make:migration add_votes_to_users_table --table=users
```

بعضی از عملیات های دیتابیس ممکن است مخرب باشند و هنگامی در جداول دیتابیس داده ای وجود داشته باشد باعث از بین رفتن برخی داده ها شود برای محافظت از این خطرات از دستور `migrate` به صورت زیر استفاده کنید و در پایان آن `--force` را قرار دهید:

کد:

```
php artisan migrate --force
```

همانطور که در پست قبل دیدید از دستور `rollback` استفاده کردیم که این دستور روی آخرین عملیات migration عمل میکند. برای اینکه همه عملیات ها را `rollback` کنیم از دستور `migrate:reset` استفاده میکنیم و در صورتی که بخواهیم همه عملیات ها `rollback` و سپس دوباره اجرا شوند از `migrate:refresh` استفاده میکنیم.

کد:

```
php artisan migrate:resetphp artisan migrate:refresh
```

فرض کنید می‌خواهیم ستون ایمیل را به جدول users اضافه کنیم ابتدا فایل migration ای ایجاد و سپس در متد up آن دستور زیر را می‌نویسیم:

کد پی‌اچ‌پی:

```
Schema::table('users', function($table)
{
    $table->string('email');
});
```

در متد down هم دستورات زیر را قرار می‌دهیم:

کد پی‌اچ‌پی:

```
Schema::table('users', function($table)
{
    $table->dropColumn('email');
});
```

حالا با اجرای دستور php artisan migrate ستون مورد نظر ایجاد خواهد شد. همچنین می‌توانیم با استفاده از after آن را بعد از ستون خاصی در دیتابیس قرار دهیم و گرنه به انتهای جدول افزوده می‌شود. برای تغییر نام جدول از متد rename از کلاس Schema مانند مثال اول در زیر استفاده کنید. با استفاده از drop هم می‌توانیم جدولی را حذف کنیم و dropIfExists هم ابتدا بررسی میکند اگر جدول وجود داشت آن را حذف میکند.

کد پی‌اچ‌پی:

```
Schema::rename($from, $to);
Schema::drop('users');

Schema::dropIfExists('users');
```

برای ویرایش یک ستون ابتدا بایستی مطمئن شوید که وابستگی doctrine/dbal روی فریمورک نصب شده باشد در غیر اینصورت مانند زیر عمل کنید:  
در بخش require فایل composer.json آن را اضافه کرده:

کد پی‌اچ‌پی:

```
"require": {
    "laravel/framework": "5.0.*",
    "illuminate/html": "~5.0",
    "doctrine/dbal": "2.5.*",
},
```

سپس از دستور composer update در ترمینال برای نصب این وابستگی استفاده کنید. همانند مثال زیر می‌توانیم یک ستون را ویرایش کنیم. در مثال زیر طول ستون name را به ۵۰ کاراکتر تغییر داده و آن را قابل NULL بودن تعریف می‌کنیم:

کد پی اچ پی:

```
Schema::table('users', function($table)
{
    $table->string('name', 50)->nullable()->change();
});
```

برای افزودن کلید خارجی به یک جدول هم به این صورت عمل میکنیم:

کد پی اچ پی:

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')->on('users');
```

ابتدا یک ستون unsigned عددی به نام user\_id ایجاد کردیم و سپس در پایین آن را کلید خارجی تعریف کردیم و گفتیم که مرجعی از ستون id از جدول users می باشد. همچنین می توانیم خاصیت onDelete آن را هم تعیین کنیم:

کد پی اچ پی:

```
$table->foreign('user_id')
->references('id')->on('users')
->onDelete('cascade');
```

برای حذف کلید خارجی هم همانند مثال زیر عمل میکنیم:

کد پی اچ پی:

```
$table->dropForeign('posts_user_id_foreign');
```

با استفاده از متد hasTable می توانیم بررسی کنیم آیا جدول موردنظر وجود دارد یا خیر و یا با استفاده از متد hasColumn بررسی کنیم در جدول موردنظر ستون های موردنظرمان وجود دارد یا خیر:

کد پی اچ پی:

```
if (Schema::hasTable('users'))
{
    //
}

if (Schema::hasColumn('users', 'email'))
{
    //
}
```

برای افزودن و حذف ایندکس می توانیم همانند مثال های زیر عمل کنیم:

کد پی اچ پی:

```
$table->string('email')->unique();
$table->dropUnique('email');
```

برای حذف ستونهای زمانی هم مانند مثال های زیر عمل کنید:

کد پی ایچ پی:

```
$table->dropTimestamps();
$table->dropSoftDeletes();
```

همچنین می توانیم موتور ذخیره سازی دیتابیس را هم مشخص کنیم:

کد پی ایچ پی:

```
$table->engine = 'InnoDB';
```

## Hash در لاراوول ۵

در لاراوول ۵ با استفاده از کلاس Hash می توانیم یک رشته را به صورت هش در بیاوریم که مناسب برای هش کردن کلمه عبور کاربران برای ذخیره در دیتابیس می باشد.

کار با متدهای Hash بسیار راحت و اندک است و نیاز به وقت زیادی برای مرور ندارد.

از متد make برای هش کردن کلمه عبور استفاده میکنیم و کلمه عبور هش شده را در دیتابیس ذخیره کنیم:

کد پی ایچ پی:

```
$password = Hash::make('secret');
```

همچنین می توانیم از تابع کمکی bcrypt نیز استفاده کنیم:

کد پی ایچ پی:

```
$password = bcrypt('secret');
```

همچنین برای بررسی صحت کلمه عبور وارد شده توسط کاربر با کلمه عبور هش شده ذخیره در دیتابیس به این صورت عمل میکنیم:

کد پی ایچ پی:

```
if (Hash::check('secret', $hashedPassword))
{
    // The passwords match...
}
```

بررسی کنیم که کلمه یا رمز مورد نظر Hash شده است یا نیاز به هش کردن مجدد دارد:

کد پی ایچ پی:

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

## Authentication در لاراوول ۵

پیاده سازی احراز هویت در لاراوول بسیار ساده است. تنظیمات مربوط به احراز هویت در پوشه config و فایل auth.php قرار دارد. در این فایل می توانید درایور را eloquent تعیین کنید و کلاس مدلی که به جدول کاربران دسترسی دارد را مشخص کنید و همچنین در بخش table نام جدولی که اطلاعات کاربران در آن ذخیره می شود را مشخص کنید.

به طور پیش فرض در پوشه app یک مدل به نام User وجود دارد که با استفاده از eloquent لاراوول یک سیستم احراز هویت را پیاده سازی کرده است که شما می توانید از آن استفاده کنید. در مسیر app/Http/Controllers/Auth دو کنترلر برای استفاده در سیستم احراز هویت استفاده می شوند که AuthController برای ایجاد کاربر جدید یا لاگین کردن و PasswordController برای ریست کردن کلمه عبور کاربرانی که آن را فراموش کرده اند به کار می رود. تمام view های مربوطه هم در پوشه resources/views/auth قرار دارند و شما می توانید آنها را به دلخواه خودتان ویرایش کنید. همچنین اگر نیاز دارید تغییراتی در فرم ثبت نام کاربر جدید بدهید کافی است در مسیر App\Services در فایل Registrar.php تغییرات مورد نظر را اعمال کنید. در متد validator می توانید قوانین اعتبارسنجی فیلدها و در متد create نیز مقادیر فیلدها را در دیتابیس و جدول users درج کنید. شما به راحتی می توانید از این سیستم احراز هویت پیش فرض لاراوول استفاده کنید.

نکته: اگر خودتان می خواهید یک migration و یا schema درست کنید و فیلدهای جدول یوزر رو بسازید حتما یادتون باشه که فیلد پسورد باید حداقل ۶۰ کاراکتر باشه و فیلد دیگری به نام remember\_token به صورت string و قابلیت نال بودن و همچنین طول ۱۰۰ کاراکتر برای بازیابی رمز فراموش شده کاربران درست نمایید.

کد پی ای پی:

```
$table->rememberToken();
```

خود لاراوول یک middleware به نام Authenticate ایجاد کرده که در متد handle آن ابتدا بررسی میکند آیا کاربر لاگین کرده یا خیر و در غیر اینصورت آن را به صفحه login هدایت میکند. شما با استفاده از این middleware در سازنده کلاس کنترلری که میخواهید فقط کاربران احراز هویت شده دسترسی داشته باشند به صورت زیر عمل کنیم. به طور مثال در کلاس کنترلر HomeController به همین صورت عمل شده است:

کد پی ای پی:

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

در صورتی که نمیخواهید از این سیستم احراز هویت تهیه شده توسط لارا اول استفاده کنید نگران نباشید. خودتان هم می توانید به سادگی آن را پیاده سازی کنید.

برای اینکار باید کلاس Auth را به کنترلر ایمپورت کنید و سپس با استفاده از متد attempt به عنوان پارامتر یک آرایه دریافت می کند و کلیدهای این آرایه نام ستون های موردنظر در دیتابیس و جدول users و مقادیر آن هم مقداروارد شده توسط کاربر است صحت اطلاعات کاربر را بررسی کنید. متد attempt در صورتی که احراز هویت با موفقیت انجام شود true و گرنه false برمیگرداند.

کد پی ایچ پی:

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

use Auth;
use Illuminate\Routing\Controller;

class AuthController extends Controller {

    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password]))
        {
            return redirect()->intended('dashboard');
        }
    }
}
```

همچنین می توانیم در متد attempt اطلاعات بیشتری را بررسی کنیم. مثلا در مثال زیر علاوه بر ایمیل و کلمه عبور باید کاربر فیلد تایید آن در دیتابیس هم ۱ باشد:

کد پی ایچ پی:

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1]))
{
    // The user is active, not suspended, and exists.
}
```

در هر قسمت از برنامه هم که نیاز دارید بررسی کنید کاربر جاری احراز هویت شده است یا خیر کافی است از متد check اینکار را انجام دهید:

کد پی ایچ پی:

```
if (Auth::check())
{
    // The user is logged in...
}
```



کد پی اچ پی:

```
if (Auth::validate($credentials))
{
    //
}
```

## لاگین کردن کاربر فقط برای یک درخواست:

می توانیم عمل لاگین کردن را فقط برای یک درخواست انجام دهیم. وقتی به درخواست مورد نظر پاسخ داده شد کاربر به صورت اتوماتیک Logout می شود. در این روش هیچ sessions و cookies ذخیره نمی شود.

کد پی اچ پی:

```
if (Auth::once($credentials))
{
    //
}
```

## Authentication Events

در مورد event ها دو مورد مهم هست .

وقتی شما از متد attempt استفاده می کنید و آن را صدا می زنید event ای به نام auth.attempt رخ خواهد داد و وقتی authentication با موفقیت انجام بشه و کاربر لاگین کنه event ای به نام auth.login رخ خواهد داد.

## آپلود فایل در لاراول ۵

الانوبت این است که با یک مثال کاربردی نحوه آپلود فایل در لاراول رو کار کنیم. فرض کنید می خواهیم در جدول posts یک مطلب جدید را اضافه کنیم که این مطلب دارای یک تصویر هم می باشد که قرار است آن را در مسیر public/uploads ذخیره کنیم. فرض میکنیم در جدول posts ستونهای (id, title, body, pic\_name) وجود دارد.

یک فایل view به نام form.blade.php در مسیر resources/views ایجاد کنید و کدهای زیر را داخل آن قرار دهید:



```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>form validation</title>
  <style type="text/css">
    .error {
      color: red;
      font-weight: bold;
    }
    .success {
      color: green;
      font-weight: bold;
    }
  </style>
</head>
<body>
  <form action="{{ url('add-post') }}" method="post" enctype="multipart/form-
data">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
    <label for="title">Title</label>
    <input type="text" name="title" id="title" value="{{ old('title') }}">
    <span class="error">{{ $errors->first('title') }}</span><br>

    <label for="post">Post</label>
    <textarea name="post" id="post">{{ old('post') }}</textarea>
    <span class="error">{{ $errors->first('post') }}</span><br>

    <label for="photo">Select an Image:</label>
    <input type="file" name="photo" id="photo">
    <span class="error">{{ $errors->first('photo') }}</span><br>

    <input type="submit" value="Submit">
  </form>
  <p class="success">{{ session('message') }}</p>
  <p class="error">{{ session('error') }}</p>
</body>
</html>

```

اکنون مسیر های زیر را در فایل routes.php تعریف میکنیم:

```

Route::get('add-post', 'PostController@getAddPost');
Route::post('add-post', 'PostController@postAddPost');

```

همانطور که می بینید باید یک کنترلر به نام PostController داشته باشیم و متدهای getAddPost و postAddPost را داخل آن تعریف کنیم.

ابتدا برای رندر کردن فایل ویو متد getAddPost را به صورت زیر بنویسید:

کد پی اچ پی:

```
public function getAddPost()
{
    return view('form');
}
```

کدهای زیر را هم در متد postAddPost قرار دهید:

کد پی اچ پی:

```
public function postAddPost(Request $request)
{
    $rules = [
        'title' => 'required|max:255|unique:posts',
        'post' => 'required',
        'photo' => 'required|image|max:1024',
    ];
    $v = Validator::make($request->all(), $rules);
    if($v->fails()){

        return redirect()->back()->withErrors($v->errors())->withInput($request-
>except('photo'));

    } else {

        $file = $request->file('photo');
        if($file->isValid()){
            $fileName = time().'_' . $file->getClientOriginalName();
            $destinationPath = public_path(). '/uploads';
            $file->move($destinationPath, $fileName);
            $post = new Post;
            $post->title = $request->input('title');
            $post->body = $request->input('post');
            $post->pic_name = $fileName;
            $post->save();

            return redirect()->back()-
>with('message', 'The post successfully inserted. ');
        } else {
            return redirect()->back()-
>with('error', 'uploaded file is not valid. ');
        }
    }
}
```

همانطور که که می بینید ابتدا مقادیر فرم را اعتبارسنجی کردیم. برای فایل هم با قانون max مشخص کردم که فایل فقط می تواند ۱۰۲۴ کیلوبایت ساینز داشته باشد و همچنین با قانون image مشخص میکنیم که فایل از نوع تصویر باشد فقط mime type های (jpeg, png, bmp, gif, or svg) را قبول میکند. در صورتی که می خواهید محدودیت بیشتری برای mime type فایل در نظر بگیرید یا اصلا فایل شما تصویر نیست می توانید با استفاده از قانون [\[url=http://laravel.com/docs/5.0/validation#rule-mimes\]\[url\]mime](http://laravel.com/docs/5.0/validation#rule-mimes) نوع فایل را مشخص کنید. در صورتی که اعتبارسنجی دارای خطا باشد به فرم برگشته و خطاها نمایش داده می شوند .

سپس اطلاعات فایل رو در متغیر file\$ قرار دادم و با استفاده از متدهای کلاس UploadedFile می توانیم به اطلاعات فایل دسترسی داشته باشیم. نام فایل را تلفیقی از timestamp جاری و نام اصلی فایل تعیین کردم تا احتمال اینکه نام فایل تکراری باشد وجود نداشته باشد و داخل متغیر fileName\$ قرار دادم. مسیر آپلود فایل را در destinationPath\$ قرار دادم و با استفاده از متد move فایل را آپلود میکنیم. این متد مسیر آپلود و نام فایل را به عنوان پارامتر میگیرد.

در نهایت سایر مقادیر فرم به همراه نام فایل را در جدول posts درج میکنیم. در صورت موفقیت یا عدم موفقیت نیز پیغام های خطایی را ست و در ویو چاپ میکنیم.

اکنون هر قسمت از وبسایت که می خواهیم پست ها را نمایش دهیم به راحتی می توانیم تصویر را هم با استفاده از نام آن نمایش دهیم:

کد پی اچ پی:

```
pic_name }" >
```

## ارسال ایمیل در لاراوول ۵

در لاراوول ۵ شما به راحتی می توانید با استفاده از کلاس Facade Mail یک ایمیل را ارسال کنید. توی این بخش هم میخوام به صورت کاربردی نحوه ارسال ایمیل را برایتان توضیح بدهم. فرض میکنیم یک فرم تماس با ما داریم که میخوایم بعد از تکمیل ان توسط کاربر به ایمیل مدیر سایت ارسال شود.

ابتدا باید در فایل env. تنظیمات مربوط به ایمیل هاست خود را ست کنید. در این مثال من تنظیمات جیمیل خودم را قرار دادم:

کد:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=*****@gmail.com
MAIL_PASSWORD=*****
```

همچنین در پوشه config و فایل mail.php هم می توانید تنظیمات بیشتری را اعمال کنید. حالا دوتا مسیر توی فایل routes.php ایجاد میکنیم:

کد پی اچ پی:

```
Route::get('contact-me', ['as' => 'contact', 'uses' => 'ContactController@contactForm']);
Route::post('contact-me', ['as' => 'contact_send', 'uses' => 'ContactController@contactSend']);
```

همانطور که مشاهده میکنید برای هر مسیر یک نام انتخاب کردم و همچنین به کنترلر ContactController و اکشن contactForm برای درخواست های GET و اکشن contactSend برای درخواست های POST نیاز داریم. متد contactForm را به این صورت می نویسیم:

کد پی اچ پی:

```
public function contactForm()
{
    return view('emails.contact');
}
```

همانطور که مشخص کردیم باید فرم تماس با ما را در پوشه emails و فایل contact.blade.php در مسیر resources/views ذخیره میکنیم و کدهای زیر را داخل آن قرار می دهیم:

کد پی اچ پی:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Contact</title>
</head>
<body>
<h1>Contact Me</h1>
<form action="{{ route('contact_send') }}" method="POST">
    <input type="hidden" value="{{ csrf_token() }}" name="_token">

    <label for="name">Your Name: </label>
    <input type="text" name="name" id="name" value="{{ old('name') }}">

    <span class="error">{{ $errors->first('name') }}</span> <br>

    <label for="email">Your Email: </label>
    <input type="email" name="email" id="email" value="{{ old('email') }}">

    <span class="error">{{ $errors->first('email') }}</span> <br>

    <label for="message">Message:</label>
    <textarea name="message" id="message">{{ old('message') }}</textarea>

    <span class="error">{{ $errors->first('message') }}</span> <br>

    <input type="submit" value="Send">
</form>

@if (Session::has('message'))
    {{ Session::get('message') }}
@endif
</body>
</html>
```

توی این مثال از ویژگی کلاس Request هم برای اعتبارسنجی استفاده میکنیم. پس با دستور زیر یک کلاس Request ایجاد میکنیم:

کد:

```
php artisan make:request ContactFormRequest
```

این کلاس در مسیر `app/Http/Requests` ایجاد می شود. آن را باز کرده و در متد `rules` آن قوانین اعتبارسنجی فرمتان را تعیین کنید.

#### کد پی اچ پی:

```
public function rules()
{
    return [
        'name' => 'required',
        'email' => 'required|email',
        'message' => 'required',
    ];
}
```

همچنین متد `authorize` را که به طور پیش فرض `false` برمیگرداند `true` کنید چون نیازی به اهراز هویت در این درخواست نداریم. خب با این کار دیگه نیازی نیست تو کنترلر اعتبارسنجی انجام بدیم فقط کافیه این کلاسی که ساختیم رو به عنوان پارامتر به متد `contactSend` بدهیم:

#### کد پی اچ پی:

```
public function contactSend(ContactFormRequest $request)
{
    extract($request->all());

    Mail::send('emails.email',
        array(
            'name' => $name,
            'email' => $email,
            'content' => $message
        ), function($message) use($email, $name) {

            $message->from($email, $name);
            $message->to('example@gmail.com')->subject('Test Email');
        });

    return Redirect::route('contact')->with('message', 'Thanks for contacting us!');
}
```

به این متد فقط درخواست های اعتبارسنجی شده وارد می شوند و اعتبارسنجی داخل کلاس `ContactFormRequest` انجام می شود. ابتدا همه داده های فرم که به صورت آرایه هست را با دستور `extract` تبدیل به متغیر کردم و با استفاده از کلاس `Mail` و متد `send` ایمیل را ارسال میکنیم. متد `send` سه تا پارامتر میگیره که اولی یک فایل وبو هست که داخل آن محتویات `html` برای ارسال فرم را تولید میکنیم و پارامتر دوم داده هایی که نیاز داریم به آن فابل ویو ارسال کنیم را در قالب آرایه میفرستیم و در پارامتر سوم هم یک `[url=http://php.net/manual/en/functions.anonymous.php][url]` تابع بی نام ایجاد

کرده و اطلاعات فرستنده و گیرنده نامه را تعیین میکنیم. در متد from نام و ایمیل فرستنده و در متد to ایمیل گیرنده نامه و در متد subject موضوع نامه را تعیین میکنیم.

در نهایت به صفحه تماس با ما ریدایرکت میکنیم و پیغامی را هم ارسال و چاپ میکنیم. همچنین باید یک فایل ویو که در متد send آن را به عنوان پارامتر اول دادیم هم در پوشه emails ایجاد کنیم. پس نام آن را email.blade.php قرار می دهیم و محتویات زیر را داخل آن می نویسیم:

کد پی اچ پی:

```
You received a message from hamo.ir:
```

```
<p>
Name: {{ $name }}
</p>

<p>
Email address :{{ $email }}
</p>

<p>
    {{ $content }}
</p>
```

## Reset Password در لارا اول ۵

تو این قسمت از آموزش به ریست کردن رمز عبور در لارا اول ۵ می پردازیم. ابتدا در مسیر app/Http/Controllers/Auth PasswordController کلاس را باز کنید و به آن متد getEmail

را اضافه کنید:

کد پی اچ پی:

```
public function getEmail()
{
    return view('auth.password');
}
```

پس بایستی یک فایل view در پوشه auth به نام password.blade.php داشته باشیم که فرم ریست کردن کلمه عبور در آن قرار دارد. محتویات این فایل شبیه زیر است:

```

@extends('app')

@section('content')
<div class="container-fluid">
  <div class="row">
    <div class="col-md-8 col-md-offset-2">
      <div class="panel panel-default">
        <div class="panel-heading">Reset Password</div>
        <div class="panel-body">
          @if (session('status'))
            <div class="alert alert-success">
              {{ session('status') }}
            </div>
          @endif

          @if (count($errors) > 0)
            <div class="alert alert-danger">
              <strong>Whoops!</strong>
              There were some problems with your input.<br><br>
              <ul>
                @foreach ($errors->all() as $error)
                  <li>{{ $error }}</li>
                @endforeach
              </ul>
            </div>
          @endif

          <form class="form-
horizontal" role="form" method="POST" action="{{ url('/password/email') }}">
            <input type="hidden" name="_token" value="{{ csrf_token()
}}">

            <div class="form-group">
              <label class="col-md-4 control-label">
                E-Mail Address
              </label>
              <div class="col-md-6">
                <input type="email" class="form-
control" name="email" value="{{ old('email') }}">
              </div>
            </div>

            <div class="form-group">
              <div class="col-md-6 col-md-offset-4">
                <button type="submit" class="btn btn-primary">
                  Send Password Reset Link
                </button>
              </div>
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
@endsection

```

این فایل از layout ای که لاراول به طور پیش فرض در پوشه views قرار داده به نام app.blade.php ارث برده می شود که از bootstrap هم استفاده میکند.

در کلاس PasswordController یک متد به نام postEmail هم برای دریافت ایمیل کاربر بعد از ارسال توسط این فرم باید ایجاد کنیم:

#### کد پی اچ پی:

```
public function postEmail(Request $request)
{
    $v = Validator::make($request->all(), [
        'email' => 'required|email|exists:users',
    ]);

    if ($v->fails())
    {
        return redirect()->back()->withErrors($v->errors());
    } else {
        $response = $this->passwords->sendResetLink($request-
>only('email'), function($m)
        {
            $m->subject($this->getEmailSubject());
        });

        switch ($response)
        {
            case PasswordBroker::RESET_LINK_SENT:
                return redirect()->back()->with('status', trans($response));

            case PasswordBroker::INVALID_USER:
                return redirect()->back()-
>withErrors(['email' => trans($response)]);
        }
    }
}
```

نکته: ابتدای کلاس کنترلر این کلاس ها را ایمپورت کنید چون در بدنه کلاس از آنها استفاده میکنیم:

#### کد پی اچ پی:

```
use Illuminate\Http\Request;
use Validator;
```

همانطور که مشاهده کردید ابتدا اعتبارسنجی رو انجام دادیم. در اعتبارسنجی هم بررسی کردیم که آیا آدرس ایمیل وارد شده در جدول users وجود دارد یا خیر بعد از ان اقدام به ارسال ایمیل به کاربر میکنیم و یک پاسخی دریافت میکنیم که این پاسخ را در حلقه switch قرار میدیم به این صورت که اگر link ریست کردن به درستی ارسال شده بود یا ایمیل کاربر نامعتبر بود به صفحه قبلی ریدایرکت شود و پیغام خطای مناسبی را در صفحه ویو چاپ کند.





بعد از اینکه کلمه عبور را تغییر دهید به طور اتوماتیک به صفحه کاربری خود ریدایرکت می شوید که این صفحه در لاراول home می باشد که می توانید با استفاده از پراپرتی redirectTo آن را تغییر دهید:

کد پی اچ پی:

```
protected $redirectTo = '/dashboard';
```

در مثال بالا آن را به مسیر dashboard تغییر دادم.

در پایان باید یادتان باشد که تنظیمات مربوط به ایمیل برنامه تان را در فایل env و config/mail.php به درستی اعمال کنید وگرنه ممکن است در ارسال ایمیل دچار خطا شوید.

## افزودن کلاس و پکیج در لاراول ۵

ممکن است شما کلاسی رو خودتون نوشته باشید و قصد دارین از اون توی فریم ورک لاراول استفاده کنید. توی لاراول ۵ به راحتی میتونید از کلاستون استفاده کنید. یک پوشه توی پوشه app به نام Classes ایجاد میکنیم و یک فایل مثلا به نام Common.php ایجاد میکنیم و کلاس Common رو داخلش تعریف میکنیم:

کد پی اچ پی:

```
<?php namespace App\Classes;

class Common
{

    public static function pre($array)
    {
        echo '<pre>' . print_r($array, true) . '</pre>';
    }
}
```

همانطور که مشاهده میکنید ابتدا یک namespace برای کلاس تعریف کردم و برای مثال داخل کلاس متدی استاتیک به نام pre تعریف کردم. حالا هر جای پروژه به راحتی می تونید به این صورت با این متد کار کنید:

کد پی اچ پی:

```
$cars = ['volvo', 'toyota', 'bmw'];
App\Classes\Common::pre($cars);
```

یا مثلاً در کنترلر بهتره اونو use کنیم و اینجوری استفاده کنیم:

کد پی اچ پی:

```
use App\Classes\Common;

class SiteController extends Controller
{
    public function index()
    {
        $cars = ['volvo', 'toyota', 'bmw'];
        return Common::pre($cars);
    }
}
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxx
```

## افزودن پکیج به لاراول

برای لاراول پکیج های زیادی نوشته میشه که شما میتونید با مراجعه به این آدرس پکیج مورد نظرتون سرچ کنید و اونو معمولاً با composer به فریمورک اضافه کنید:

<http://packalyst.com/>

مثلاً یکی از پکیج های خوبش اینه که کار با تصویر رو براتون آسون میکنه

<http://packalyst.com/packages/packag...rvention/image>

یا پکیج debug-bar لاراول که خیلی کاربردی

<http://packalyst.com/packages/packag...ravel-debugbar>

# آموزش فارسی

## AMOZESHFARSI.IR