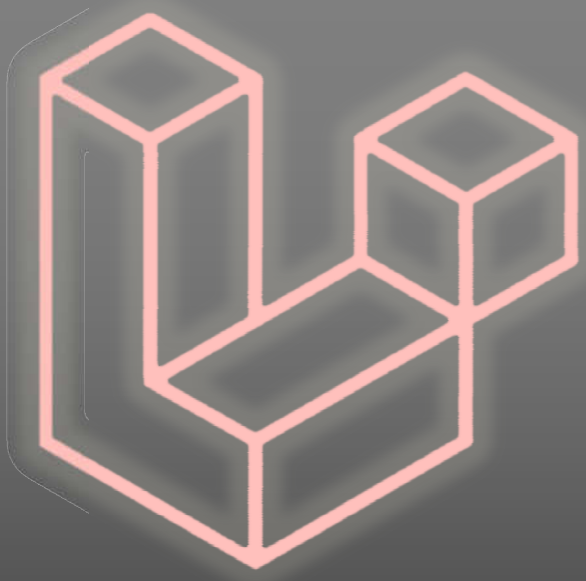
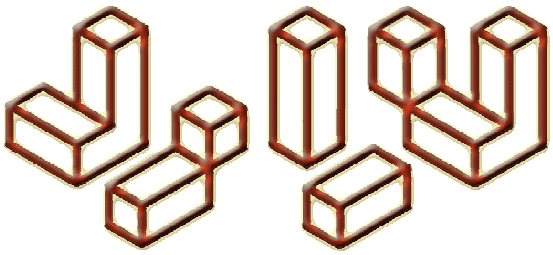
آموزش تصویری
LARAVEL

چارچوبی برای برنامه نویسی
به شکلی سریعتر در PHP



پدرام رحیمی
@pedramrahimi1975



I don’t know has become I don’t know yet! - Bill Gates

من نمیدانم

به هنوز نمیدانم

بدل شده!

بیل گیتس

فهرست

آشنایی

زبان PHP[1] یک زبان برنامه‌نویسی پیشرفته از نوع شیءگرا[2] برای ساخت برنامه‌های تحت وب[3] است که در سال ۱۹۹۵ میلادی ساخته شده است. نام آن در واقع به معنای پیش پردازنده‌ی قبل از فرامتن[4] است که روی کامپیوتر میزبان[5] یا سرویس دهنده[6] اقداماتی را انجام داده و نتیجه را همراه کدهای HTML[7] به مرورگر ارسال میکند. کارهایی مثل اتصال به بانک اطلاعاتی[8] ، دریافت داده[9] و محاسبات و الگوریتم های جستجو، که از صفحات معمولی وب بر نمی آیند. پیش‌نیاز فراگیری این زبان هم، آشنایی با ساخت صفحات وب از طریق کدهای HTML است.

هر چند برنامه‌نویسانی که پیش از این توسعه‌ی برنامه‌های خود را از طریق محصولات شرکت مایکروسافت پیش برده بودند، یک سال بعد ابزاری به نام ASP[10] را در اختیار داشتند. اما افرادی هم که قبل از آن با محیط های رایگان با منبع باز[11] در این عرصه مشغول بوده‌اند، وفاداری خود را به این ابزارها حفظ کرده و تا امروز با PHP کار کرده اند. با توجه به اینکه این زبان به نسخه‌های جدیدتر هم به‌روزرسانی می‌شود، می‌توان نتیجه گرفت که هیچ کمبودی در آن نسبت به فنّاوری‌های دیگر که در انحصار شرکت های بزرگی مثل مایکروسافت هستند، وجود ندارد. در عین حال سابقه‌دار بودن این زبان باعث شده است که مزیت‌های ویژه‌ای هم نسبت به تکنولوژی‌های مشابه داشته باشد. از جمله:

- امکان اجرای آن روی بیشتر سیستم‌عامل‌های کامپیوتر رومیزی یا تلفن‌های هوشمند!
- وجود بیشترین کاربران، انجمن‌های آنلاین و اسناد و متون راهنما برای یادگیری.
- زبان برنامه‌نویسی مهم‌ترین سایت‌های دنیا مثل ویکی‌پدیا یا فیس‌بوک.
- و انحصاری نبودن تمام مراحل توسعه از ویرایشگر تا محیط اجرا.

که همه بر محبوبیت این زبان افزوده‌اند.

اما اصرار محیط‌های کار تیمی با تأکیدی که برای تعیین ساختارهای منظم در کدنویسی و نوشتن کدهای تمیز و توضیح سرخود[12] داشتند، منجر به خلق معماری‌های ویژه‌ای برای تولید نرم‌افزار شد.

---

[1] Pre Hypertext Processor
[2] Object oriented
[3] Web applications
[4] Heypertext
[5] Host
[6] Server
[7] Hyper Text Markup Language
[8] Database
[9] Data
[10] Active Server Page
[11] Open source
[12] Self-document

یکی از پرطرفدارترین این ساختارها MVC[۱۳] بود. به طور خلاصه هدف از این معماری یا شیوه‌ی کدنویسی این بود که کدهای یک برنامه‌ی مرتبط با بانک اطلاعاتی، در سه بخش مجزا نوشته شوند. به شکلی که بخشی از این کدها کار **کنترل** محاسبات و درخواست، بخش دیگر یعنی **مدل** ارتباط با بانک اطلاعاتی و بازگرداندن پاسخ مناسب به بخش کنترل و در نهایت بخش سوم کار **نمایش**[۱۴] نتیجه پس از گردش کار در بخشهای کنترل و مدل را بر عهده داشته باشد.

هر چند این شیوه نظم و ساختار باارزشی برای فهم کدها و انجام ساده‌تر پروژه‌های تیمی ایجاد کرد، اما فراگیری آن هم به شکل دقیق برای پیاده‌سازی و تبعیت از این شیوه آسان نبود. به عنوان مثال برنامه نویسان یا فراموش می‌کردند و یا دقیق نمی‌دانستند که کدام کدها را باید در کدام یک از سه بخش مدل، نمایش و کنترل بنویسند!

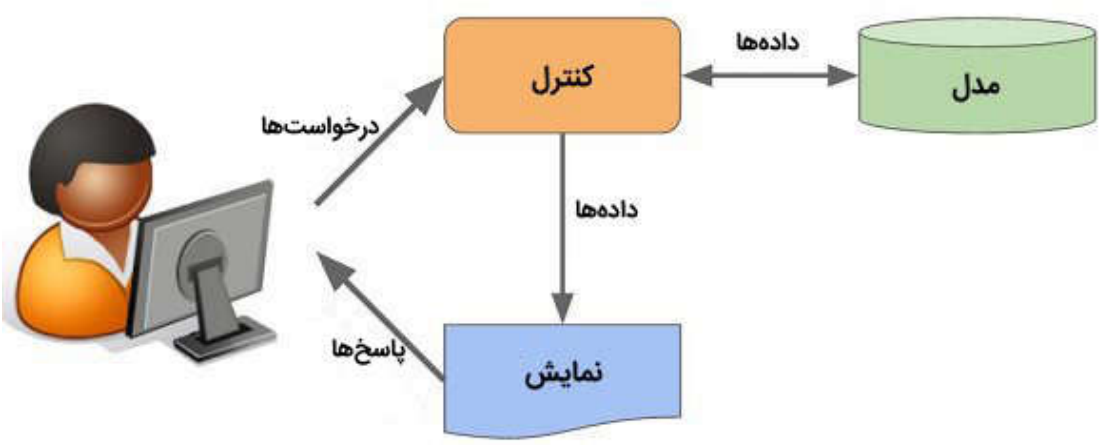
در سال ۲۰۱۱ میلادی آقای تیلور آتوِل[۱۵] سرانجام با نگارش یک مجموعه کد و دستور که به عنوان چارچوب[۱۶] برای نگارش برنامه‌های PHP تهیه کرده بود، راه‌حلی ارزشمند را در تولید برنامه با معماری MVC ارائه کرد. به گونه‌ای که با دستورات ویژه‌ی این کتابخانه ی کامل کد، که نامش لاراول[۱۷] بود، از این پس امکان نگارش منظم کد در استانداردی ویژه و سرعت بالا مقدور بود. یعنی برنامه‌نویس گاهی به جای نگارش دهها و صدها خط کد تنها با یک دستور اقدام به پیاده‌سازی صفحات و بخش‌های مورد نظر خود میکرد. علاوه بر آن روند کار به گونه‌ای بود که کدها به طور خودکار در بخش‌های مجزا و مرتبط با معماری MVC هم قرار می‌گرفتند!

لاراول معنای خاصی ندارد و به گفته ی مخترع آن از نام قلعه ای در فیلم سینمایی نارنیا[۱۸] به نام پاراول گرفته شده است! اما با این حال توانسته لقب بهترین فریم ورک PHP را به خود اختصاص دهد. در واقع زبان PHP به همراه این چهل مگابایت کدی که شما به پروژه ی خود اضافه می‌کنید، چنان قدرتی پیدا می‌کند که می‌توان گفت دوباره در ظاهری بسیار کامل‌تر متولد شده است.

مزایای لاراول به همین چند خط توضیحات محدود نمی شود. به زودی با مطالعه ی این کتاب متوجه خواهید شد که چطور این کتابخانه ی ارزشمند توانسته علاوه بر موارد ذکر شده، امنیت ویژه ای را برای جلوگیری از نفوذ هکرها نیز در برنامه های شما فراهم کند و به حق تبدیل به یک استاندارد کدنویسی جادویی گردد.

[13] Model View Control
[14] View
[15] Taylor Otwell
[16] Framework
[17] Laravel
[18] The Chronicles of Narnia

توصیه می شود برای یادگیری مطالب، حتماً خط به خط آن را شخصاً اجرا کرده و با نتایج ذکر شده در اینجا مقایسه کنید.



نصب و راه‌اندازی

روشن است که نصب لاراول را باید با نصب PHP آغاز کرد. اما نسخه ی نصب شده باید سازگاری لازم را داشته باشند. در ابتدا به سایت لاراول به نشانی زیر مراجعه کنید:

www.laravel.com

در این سایت به بخش **Documents** یا اسناد که مراجعه کنید، نسبت به اینکه کدام نسخه از لاراول را خواسته باشید (و البته به‌طورپیش‌فرض آخرین نسخه) که از گوشه‌ی بالای سایت در بخش Version قابل انتخاب است، تمام نیازمندی‌های نرم‌افزاری با عنوان Server Requirements معرفی شده‌اند. شک نیست با توجه به تعدد و پیچیدگی این تنظیمات و همچنین نیازی که به نصب بانک اطلاعاتی پیش‌فرض و سازگار با PHP یعنی My-SQL هم داریم، باید روشی برای نصب سریع و آسان و همچنین تنظیمات لازم هم وجود داشته باشد. چون راه‌اندازی جداگانه‌ی هر کدام از این سرویس‌ها یا تنظیمات ممکن است مشکل یا وقت‌گیر باشد.

نصب PHP و My-SQL

برنامه نویسان حرفه‌ای به‌طور معمول روش دیگری برای این کار دارند و آن نصب برنامه‌ی زمپ از نشانی زیر است:

www.apachefriends.org

بعد از مراجعه به سایت بالا بسته به اینکه از چه سیستم‌عاملی استفاده می‌کنید، نسخه‌ی مناسب به شما توصیه می‌شود. معمولاً نسخه‌ی مترجم زبان[۱۹] PHP روی این برنامه بالاتر از نیاز لاراول است. بنابراین با خیال راحت برنامه‌ی XAMPP را دریافت و نصب کرده و مطمئن باشید که آخرین نسخه‌ی لاراول هم به‌طور کامل در آن اجرا خواهد شد.

در ویندوز که نصب بسیار ساده و با زدن مکرر دکمه‌های Next انجام خواهد شد. دستور نصب در سیستم‌عامل لینوکس را هم حرفه‌ای‌ها کافی است که یک بار از سایت زمپ یا منابع و انجمن‌های مرتبط خوانده و یاد بگیرند.

برنامه‌ی سرشناس XAMPP مجموعه‌ی کاملی است که یک وبسرور تنظیم شده با قابلیت تفسیر کدهای PHP و ساخت و مدیریت بانک‌های اطلاعاتی MySQL را از طریق یک وب اپلیکیشن به نام phpMyAdmin فراهم می‌کند. در واقع پنل کاربری شرکت‌های اجاره‌ی فضای اینترنتی (Hosting) هم چیزی شبیه همین نرم‌افزار هستند که برای نصب برنامه‌های تحت وب روی اینترنت به کار گرفته می‌شوند.

بعد از نصب زمپ در ویندوز، مسیر زیر محل قرارگیری فایل‌های پروژه‌ی شما خواهد بود:

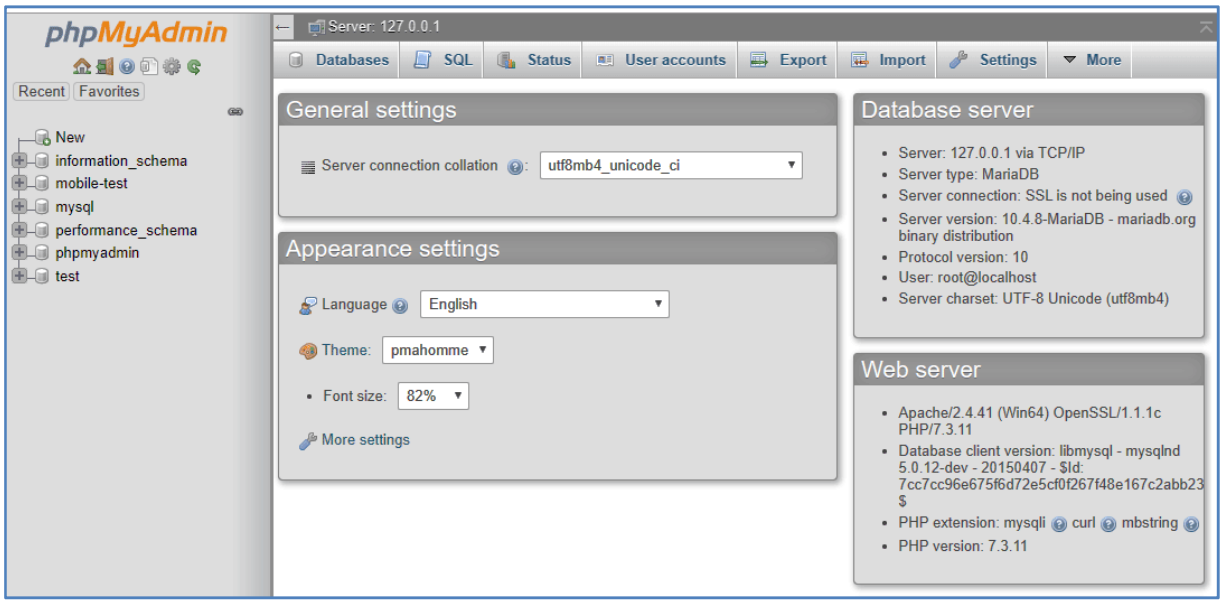
C : \ xampp \ htdocs

کافی است در اینجا یک پوشه به نام php-projects بسازید و فایل‌های خود را از این به بعد در آن کپی کنید. حالا برای اجرای آن‌ها نام پوشه و نام فایل مورد نظر را بعد از عبارت زیر در مرورگر تایپ کنید:

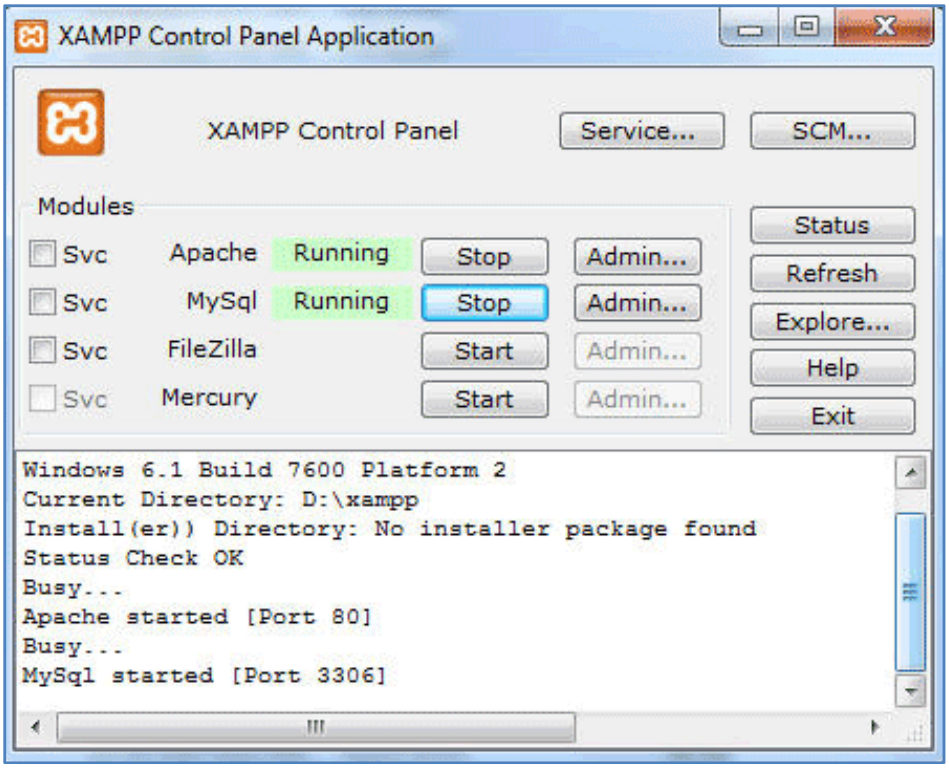
http : / / localhost

با ورود عبارت بالا به تنهایی در مرورگر باید صفحه‌ی خوش‌آمدگویی زمپ را ببینید. در منوی بالای آن روی عبارت phpMyAdmin کلیک کنید تا تصویر زیر ظاهر شود:

[19] Compiler



دیدن تصویر بالا علامت آن است که همه چیز آماده‌ی کار بوده و شما قادر به اجرای کدهای خود هستید. اگر به هر دلیلی چیزی اجرا نشد، فایل کنترل پنل زمپ را که در ویندوز معمولاً کنار ساعت و گوشه‌ی دسکتاپ قرار گرفته باز کنید و سرویس‌های آن را به‌صورت زیر بررسی کنید:



اگر سرویس Apache در حالت اجرا (Running) نبود باید Start را بزنید. گاهی به دلیل نصب وب سرورهای دیگر روی سیستم‌عامل، تداخل ایجاد شده و این سرویس ران نمی‌شود. راه حل آن

است که پورت اجرای زمپ را تغییر دهید. برای این منظور در پوشه‌ی زمپ فایل httpd.conf را توسط Notepad یا هر ویرایشگر متنی، باز کرده و عبارت Listen 80 را در آن جستجو کنید. سپس این عدد را مثلاً به 10 تغییر دهید. حالا سرویس آپاچ را مجدداً اجرا کنید و این بار با آدرس localhost:10 می‌توانید در مرورگر به زمپ متصل شوید.

برای تست اولین برنامه‌ی شخصی هم کافی است یک ویرایشگر فایل متنی مثل Notepad یا هر ویرایشگر دیگری را باز کرده و در فایلی با نام index.php در مسیر زیر:

C:\xampp\htdocs\php-projects

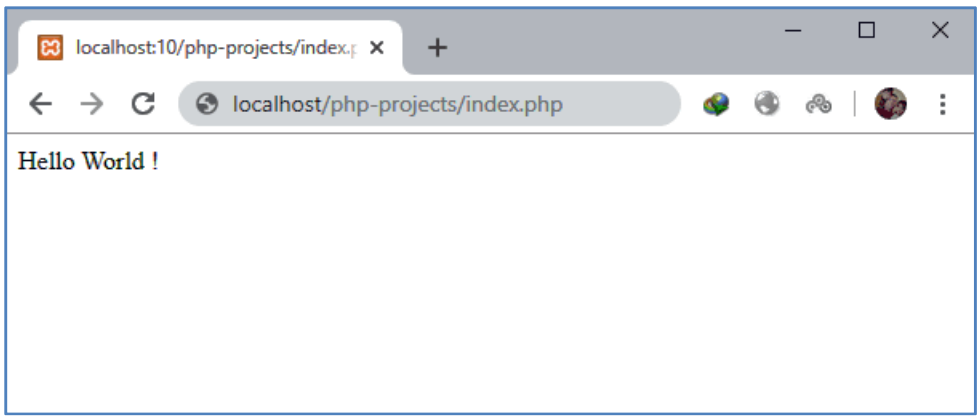
دستورات زیر را وارد کنید:

```
<?php
echo "Hello World !" ;
?>
```

فعلاً نیازی به توضیح طولانی در مورد دستورات بالا ندارید. فقط بدانید که دستورات پی اچ پی وسط دو بخش <? و php?> قرار می گیرند. فایل بالا را ذخیره کنید و نشانی زیر را در مرورگر اجرا کنید:

http://localhost/php-projects/index.php

نتیجه باید به صورت زیر دیده شود:



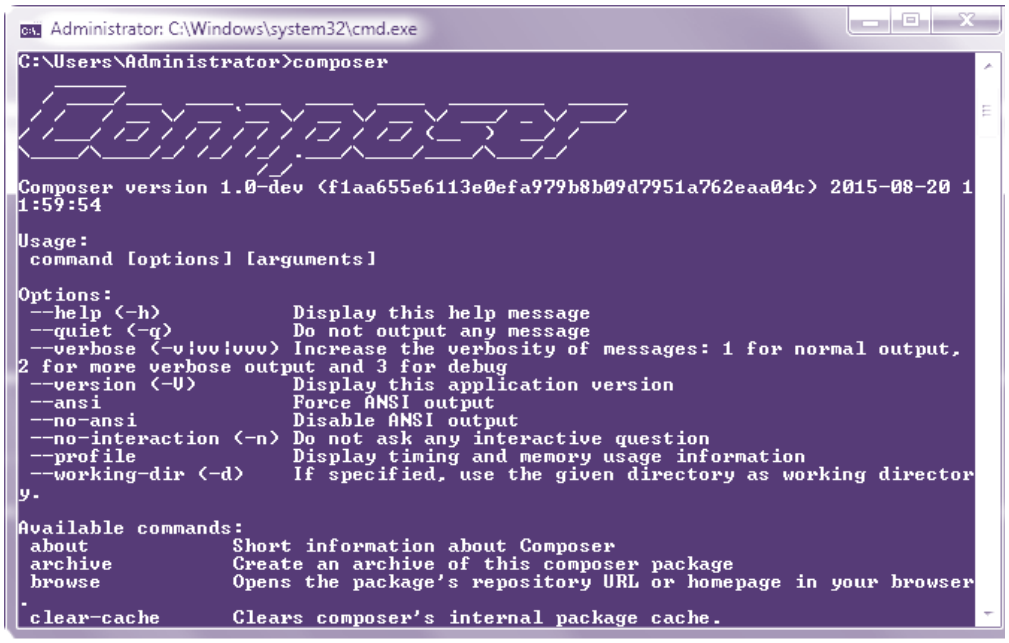
نصب Laravel

راههای متعددی برای این کار وجود دارند. به‌عنوان سریع‌ترین روش با نصب برنامه‌ی **Composer** از نشانی زیر شروع کنید:

www.getcomposer.org

در واقع یک مدیریت دانلود را برای پکیج‌های PHP نصب کرده‌اید. در ویندوز Composer-Setup.exe به راحتی نصب خواهد شد. البته شما هیچ برنامه‌ی خاصی در فهرست برنامه‌های نصب شده نخواهید دید. درواقع این برنامه از طریق خط فرمان کار خواهد کرد. اما برای نصب در لینوکس هم در منوی Download داخل همین سایت دستورات لازم برای اجرا در ترمینال نوشته شده‌اند.

پس از نصب کافی است یک پنجره‌ی خط فرمان[۲۰] باز کنید و در آن دستور **composer** را مستقیم تایپ کنید تا راهنمای دستور و سوئیچ‌های آن را مشاهده کنید:



حالا ابتدایی‌ترین روش نصب لاراول با دستور زیر خواهد بود:

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

[20] Command prompt

اگر دستور بالا را در هر مسیری که کامندپرامپت در آن قرار دارد اجرا کنید، پوشه‌ای به نام blog (این نام به‌عنوان آخرین بخش دستور، انتخابی است) ساخته شده و بسته به سرعت اینترنت شما، فایل‌های مورد نیاز پروژه در حدود ۴۰ مگابایت بعد از چند دقیقه در این مسیر ذخیره می‌شوند. توجه داشته باشید که قبل از هر ویرایشی روی این فایل‌ها **می‌توانید یک نسخه‌ی پشتیبان از این پوشه تهیه کنید تا برای پروژه‌های بعدی عیناً در مسیر و پوشه‌ای با نام جدید کپی کرده و استفاده کنید.**

روش‌های دیگر را می‌توانید از طریق بخش Installation در سایت لاراول مطالعه کنید. همچنین مخزن کد github در اینترنت نیز بخشی مربوط به Laravel دارد که اجازه ی دانلود مستقیم را به صورت یک فایل zip. میدهد.

یا اینکه می توان با استفاده از دستور زیر یک نصب عمومی برای «نصب کننده» انجام داد:

composer global require laravel/installer

بعد از این هر بار در هر پوشه ی خالی و جدیدی، دستور **laravel new** را در خط فرمان تایپ کنید، از این منبع که در مسیر عمومی سیستم عامل به صورت فشرده[۲۱] قرار دارد، یک نسخه کپی و باز شده و آماده ی برنامه نویسی خواهد بود.

حالا برای تست، پوشه‌ی فایل‌های پروژه را در مسیر اجرای فایل‌های php یعنی پوشه‌ی htdocs در محل نصب زمپ کپی کنید. نشانی آن اگر همان نام blog را داشته باشد به صورت زیر خواهد بود:

C:\xampp\htdocs\blog

برای اجرا دو راه داریم! یکی راه معمول اجرای پروژه‌های php که از localhost است. فقط مسیر اولیه‌ی این پروژه public است. پس اجرای آن به صورت زیر خواهد بود:

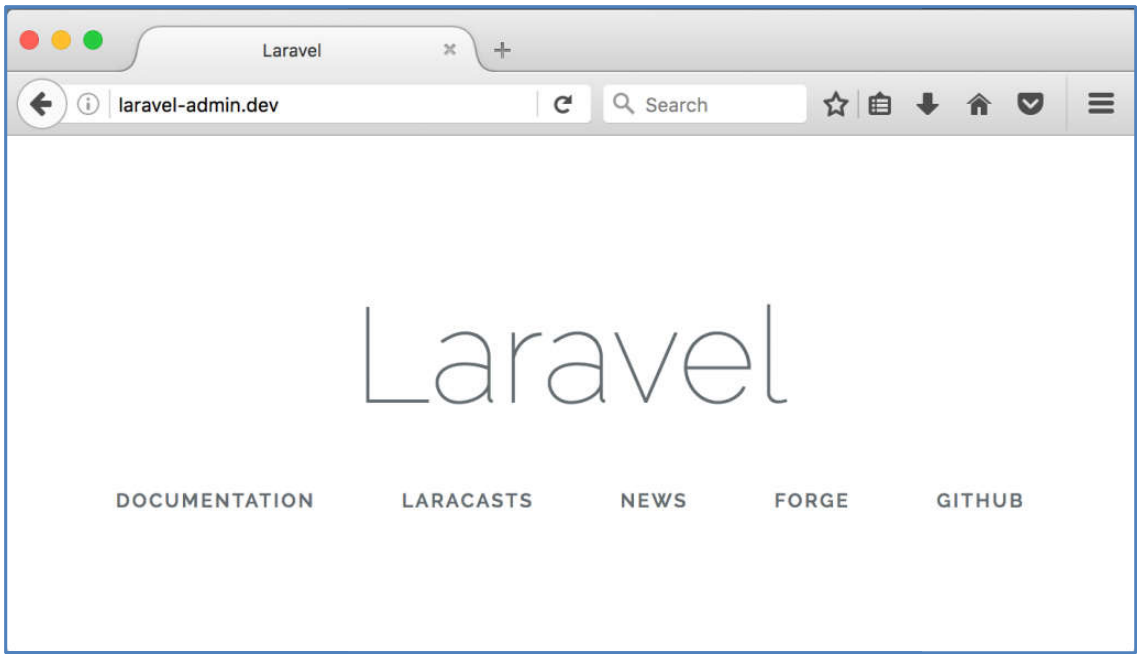
http://localhost/blog/public

اما این بدون استفاده از دستورات لاراول و به کمک برنامه‌ی زمپ است. راه دیگر آن است که سِرور ویژه‌ی لاراول را بشناسیم. برای این منظور کافی است در مسیر پروژه و از طریق خط فرمان دستور زیر را اجرا کنید:

php artisan serve

[21] Zip

بعد از چند ثانیه فایل artisan که در ریشه‌ی پروژه‌ی لاراول شما بدون هیچ پسوندی وجود دارد یک سِرور ویژه در پورت ۸۰۰۰ ایجاد می‌کند که این بار از طریق آدرس **localhost:8000** به طور اختصاصی برای پروژه‌ی شما فعال شده است! کاربردهای دیگر فایل artisan را در ادامه‌ی کار با پروژه‌های لاراول خواهید دید. اما تصویری که پس از اجرای پروژه ی خام لاراول در مرورگر[۲۲] خواهید دید به صورت زیر است:



نکته: چرا گاهی دستور PHP در کامند پرامپت اجرا نمی‌شود؟

تنها نکته‌ی باقیمانده ممکن است عدم اجرای دستور php به طور مستقیم در خط فرمان باشد. این موضوع به آدرس‌دهی‌های شما در سیستم‌عامل برمی‌گردد. برای رفع این مشکل کافی است در خط فرمان تایپ کنید:

```
set   path= c:\xampp\php
```

این حالت با بستن پنجره‌ی خط فرمان خنثا خواهد شد. برای ماندگار بودن این تعریف مسیر کافی است در ویندوز بخش Environmental variables را باز کرده (همین عبارت را در ویندوز سرچ کنید) و در بخش System variables قسمت Path را باز کرده، مسیر مورد نظر را به آن اضافه کنید. بعد از این از هر مسیری فایل php.exe را می‌توان فراخوانی کرد.

[22] browser

ویرایشگر کد

برای زبانی با اهمیت PHP ویرایشگرهای کد متعدد و قدرتمندی تولید شده‌اند که برخی رایگان و برخی خریدنی هستند. با توجه به ماهیت اوپن سورس بودن این زبان، در اینجا نیز یک ویرایشگر قدرتمند و مدرن ولی رایگان معرفی می‌شود. نصب اولیه‌ی ویژوال استودیو کد، هیچ تنظیم خاصی ندارد و مثل نصب تمام برنامه‌های دیگر است که بسته به سیستم‌عامل مورد استفاده‌ی خودتان می‌توانید نسخه‌ی مربوط را دریافت کنید. این ویرایشگر کد خودش به تنهایی چیزی در حد نوتپد است. در واقع افزونه‌ها[۲۳] هستند که بعد از آن نصب می‌شوند و بسته به زبان انتخابی شما قابلیت‌های جالبی را در هنگام تایپ کد، مثل کامل کردن دستورات[۲۴] با تایپ حرف اول آن‌ها را در اختیار می‌گذارند.

پس بعد از نصب به قسمت Extensionsدر نوار سمت چپ برنامه مراجعه کنید. از بخش جستجوی افزونه‌ها در بالا، افزونه‌ی HTML Snippets را انتخاب و با زدن Install در زیر آن نصبش کنید. همین‌طور PHP Snippets یا نظیر آن‌ها برای Laravel را هم پیدا کرده، دانلود و نصب کنید.

هر چند که به نظر می رسد بسیاری از برنامه نویسان قدیمی تر هم ابزارهایی کدنویسی PHP را مثل PHP-storm به کار بگیرند که در خروجی کار شما تأثیری نخواهد داشت.

ساختار پایه ی پروژه

در سایت لاراول بخش اسناد به دنبال Directory structure بگردید. در آنجا به خوبی محتوای تمام پوشه ها و فایل شرح داده شده اند. با توجه به اینکه هنوز شما هیچ پروژه ای نساخته اید این حدود چهل مگابایت فایل را وابستگی های[۲۵] پروژه می نامند. در زبانهای دیگر هم کم بیش چنین چیزی برقرار است. مثلاً در پروژه های داتنت شنیده اید که باید فلان پکیج دات نت فریم ورک را نصب کنید تا بتوانید دستورات خود را اجرا کنید. همانطور که به نظر می رسد وابستگی ها در اینجا بسیار مختصرند و برای سهولت کار، به خود پروژه می چسبند تا همه جا بی دردسر اجرا شوند. با این حال نکته ی مهم این است که بدانید نباید تمام پوشه ها و فایل ها را دستکاری کنید.

[23] Extensions
[24] Intellisense
[25] Dependencies

به عنوان مثال پوشه ی app هسته ی اصلی پروژه ی شما را تشکیل می دهد و فایلهای زیادی را درون آن خواهید ساخت.

یا پوشه ی config مربوط به بحث تنظیمات پروژه است.

در پوشه ی public یا عمومی فایلهای css، javascript یا فایل اولیه ی پروژه مثل index.php قرار دارد.

یکی از فایلهای مهم که متوجه شده اید در ریشه ی این مجموعه قرار دارد، فایل artisan است که دستورات مهم لاراول را در خط فرمان اجرا می کند.

فایل composer.json فهرست وابستگی های نصب شده روی پروژه فعلی را نگه می دارد.

با توجه به ساختاری که در بخش آشنایی در مورد آن صحبت شد یعنی معماری MVC شما متون یا تگ های HTML را که در صفحه ی اجرای برنامه به کار رفته اند، در فایل index.php نخواهید دید! در واقع طبق معماری یاد شده، باید در پوشه ی Resources و بخش views به دنبال آنها بگردید. اینجا جای صفحاتی است که دستورات نمایش را از بخش های کنترل و مدل گرفته و به اجرا می گذارند.

مسیردهی ها

پایه ای ترین موضوع برای شروع کدنویسی لاراول از این بخش آغاز می شود. آماده باشید که اولین کدها را بنویسید. باز در بخش اسناد لاراوال در سایت اولین چیزی که در قسمت مبانی یا Basics می بینید همین مسیردهی یا **Routing** است.

در ویرایشگر کد خود هر چه که باشد، ابتدا پوشه ی اصلی پروژه را با Open folder یا دستور مشابه آن باز کنید. اگر در ویژوال استودیو کد باشید، در نوار سمت چپ تمام فایلها و پوشه ها قابل ردیابی هستند. همانطور که در بخش قبل هم گفته شد برای کار با تمام پوشه ها کار نداریم. بیشتر پوشه های app، public، resources و routes را در نظر بگیرید.

همانطور که متوجه شده اید در مسیر resources/views فایلی به نام welcome.blade.php قرار دارد. همان فایلی که صفحه ی اولیه ی پروژه ی لاراوال را نمایش

می دهد. با خیال راحت تمام کدهای درون آن را می توانید پاک کنید و عبارات مدنظر خود را قرار دهید! حالا با اجرای مجدد پروژه مطالب جدید را خواهید دید. لاراول در موتور پوسته[۲۶] فقط فایلهایی را به صورت View در معماری MVC خواهد شناخت که یک کلمه ی blade قبل از پایان نام آنها آمده باشد. بنابراین نوشتن welcome.php مجاز نیست و باعث می شود نتوانیم دستورات لاراول را به کار ببریم.

حالا در مسیر routes فایل web.php را در ویرایشگر کد باز کنید:

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

دستوری که در اینجا می بینید نیز مانند فایل ویوی نمونه ای که دیدید برای راهنمایی و اجرای اولین صفحه ی پروژه به صورت پیشفرض نوشته شده است. یک کلاس[۲۷] به نام Route که در ابتدای خود یک عملگر[۲۸] دریافت مسیر به نام get دارد و در ادامه بعد از دریافت مسیر ریشه ی سایت '/' یک ویو به نام welcome را فراخوانده است. کافیست شما یک ویوی جدید در مسیر resources/views بسازید و نام آن را اینجا وارد کنید تا ارجاع به آن صورت بگیرد.

همچنین نمونه های دیگری را می توانید به شکل زیر آزمایش کنید:

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

یعنی یک کپی از دستور قبلی در زیر آن درست کنید و این بار نشانی را جوری قرار دهید که با localhost:8000/login فراخوانده شود. بعد فایلی هم به نام login.blade.php باید داشته باشید در بخش ویوها که توسط این نشانی فراخوانی شود.

ساخت کنترلر

همانطور که شاید متوجه شده باشید، در مثال های قبلی نقش کنترل و مدل در نظر گرفته نشده است. یعنی یک درخواست کاربرد مستقیم یک بخش ویو را فراخوانی کرده بدون آنکه محاسبه ی خاصی صورت گرفته باشد یا پارامتر به صفحه پاس گردد.

[26] Template engine
[27] Class
[28] Function

مسیر app/Http/Controllers را در پروژه باز کنید. مانند موارد قبلی یک Controller.php برای نمونه در این قسمت دیده می شود. پس یک راه شاید این باشد که مانند این نمونه کنترلی را به صورت دستی ایجاد کنیم که به صورت یک فایل جدید بتواند قبل از نمایش هر ویو از بخش مسیریابی فراخوانده شود. همانطور که در بخش آشنایی گفته شد لاراول امکاناتی دارد که مانع از درج کدهای اشتباه یا نابجا و مغایر با معماری MVC می گردد. پس اینجا بهتر است از این امکانات استفاده کرده و از ساخت مستقیم کنترلر خودداری کنیم.

فرض کنید که میخواهیم کنترلری برای بخش index یا ورودی سایت بسازیم. پس یک خط فرمان در مسیر پروژه باز کنید و دستور زیر وارد کنید:

php artisan make:controller indexController

در اینجا با کاربرد جدید از دستور artisan آشنا شدید. حالا اگر باز به مسیر app/Http/Controllers مراجعه کنید فایل جدیدی با نام indexController.php را خواهید دید. اگر این فایل را در ویرایشگر کد باز کنید متوجه ی نکته ی با اهمیتی خواهید شد. بله کدهایی را که برای ابتدای محتوای این فایل لازم داشته ایم دستور artisan به صورت خودکار ساخته است:

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class indexController extends Controller
{
    //
}
```

در اینجا کلاسی به نام indexController ساخته شده است که از کلاس Controller ارث بری دارد. حالا باید به بخش مسیرها برگردید و مسیر صفحه ی اصلی را به صورت زیر در فایل web.php به صورت زیر تغییر دهید:

```
Route::get('/', 'indexController@index');
```

یعنی به ازای مسیر ریشه ی سایت که درخواست داده میشود، برو کنترلری به نام indexController را فراخوانی کرده و درون آن به دنبال عملگری به نام index برای اجرا بگرد.

به صفحه ی indexController خود برگشته و کد زیر را در آن وارد کنید:

```
class indexController extends Controller
{
   public function index(){
```

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

ما به جای // که به صورت پیشفرض در کنترلر خالی وجود داشت یک عملگر index ساختیم که همان کار اولیه در مسیردهی را انجام داده و یک ویو به نام welcome را برگرداند. در واقع از MVC بخش VC را پیاده سازی کرده ایم. اگر دوباره پروژه را با localhost:8000 اجرا کنیم نتیجه مانند قبل خواهد بود. اما روشن است که هدف فقط این نبوده که یک ویو را در کنترلر صدا بزنیم. در واقع قرار است پردازش های دیگری را انجام دهیم و نتیجه را از طریق ویو فراخوانی کنیم.

به عنوان مثال فرض کنید بخواهیم متغیری را در کنترلر اکنون تعریف کنیم و در ویو آن را داشته باشیم:

```
public function index(){
    $data = ["name"=>"پدرام رحیمی"];
    return view('welcome',$data);
}
```

همانطور که روشن است view این بار دو پارامتر گرفته است، یکی نام ویو برای نمایش و دیگری متغیر آرایه ای که بالاتر مقدار گرفته و قرار است به ویو پاس داده شود.اگر به فایل welcome.blade.php رفته و عبارت زیر را در بدنه ی HTML وارد کنید:

```
{{$name}}
```

خواهید دید که مقدار متغیر پاس شده به خوبی در صفحه پس از اجرا نمایش داده خواهد شد. همینجا به یکی از امکانات لاراول برخوردید. تمام دستورات یا متغیرهای برنامه را می توان با گذاشتن میان {{..}} اجرا کنید. در واقع این علائم معادل <?php echo .. ?> هستند.

به آرایه ی خود مقادیر بیشتر هم می توانید بدهید:

```
$data = ["name"=>"پدرام رحیمی","age" => 45];
```

در این صورت می توانید در ویو همه ی این مقادیر را با ذکر نام متغیر چاپ کنید. این مقادیر می توانند همان اطلاعاتی باشند که می توان از مدل و بانک اطلاعات دریافت کرد.

اما ممکن است بخواهید در مقصد نام متغیر متفاوت با چیزی باشد که در کنترلر تعریف کرده ایم:

```
$text = "صفحه ی اصلی";
return view('welcome')->with('pageTitle',$text);
```

در اینجا با استفاده از دستور with توانستیم مقدار متغیر $text را به متغیر pageTitle مربوط کنیم. حالا کافیست در مقصد یعنی صفحه ی welcome داشته باشیم:

```
{{$pageTitle}}
```

که مقدار همانطور که می خواهیم نمایش داده می شود. اما زمانی که نخواهیم متغیری را تغییر دهیم و به صورت آرایه ای هم چیزی را نفرستیم به جای with از compact به روش زیر استفاده می کنیم:

```
$pageTitle = "صفحه ی اصلی";
    return view('welcome', compact('pageTitle'));
```

که متغیر را با همان نام به ویو ارسال می کند. خاطرتان هست که پارامتر دوم view بدون compact فقط یک آرایه می تواند باشد و ما compact را برای متغیرهای معمولی به کار می بریم. با این دستور می توان متغیرهای متعددی را ارسال کرد:

```
$pageTitle = "صفحه ی اصلی";
$name = "پدرام رحیمی";
    return view('welcome', compact('pageTitle','name'));
```

و این ها روش های مختلف ارسال مقدار به ویو از طریق کنترلر هستند.

مسیرهای دارای پارامتر

نکته ی مهم این است که شما مجبور نیستید برای هر صفحه یک کنترلر مجزا بسازید. همین که یک عملگر دیگر در این کنترلر تعریف کرده و در مسیردهی نام عملگر جدید را همراه همین کنترلر برای یک ویوی دیگر به کار ببرید، همه چیز درست کار خواهد کرد.

یک ویوی جدید به نام article.blade.php در مسیر views بسازید. در مسیر Routes اما دستور زیر را در web.php وارد کنید:

```
Route::get('/article/{id}', 'indexController@article');
```

می بینید که اول از همه مسیر /article یک {id} هم دریافت می کند که مثلاً می تواند شماره ی مقاله ای باشد که می خواهیم در این صفحه به نمایش بگذاریم. حالا به indexController برگشته و عملگر زیر را برای آن می نویسیم:

```
public function article($id){
    return view('article',compact('id'));
```

```
}
```

که در این صورت اگر در داخل فایل article.blade.php داشته باشیم:

```
{{$id}} شماره ی مقاله
```

و نشانی زیر را در مرورگر وارد کنیم:

Localhost:8000/article/3

خواهیم دید که شماره ی ارسال شده از طریق آدرس که همان ۳ هست، در صفحه ی article نمایش داده می شود. توجه داشته باشید که چون در بخش مسیردهی یک id تعریف شده، نشانی article به تنهایی کار نمی کند. اما عدد بعد از آدرس را هر عددی می توانید بدهید. حتا شما می توانید یک نام یا یک رشته به جای عدد هم وارد کنید.

تا همین جا با آنکه هنوز مقادیر از سمت بانک اطلاعاتی نیامده اند، روش های پاس کردن آنها را به صفحات آموخته اید. کارهایی که با PHP بسیار مشکلتر از این انجام خواهند شد. روشن است که با دانستن این بخش ها، کار با مدل ها ساده تر جلوه خواهند کرد.

بانک اطلاعات

از اینجا به بعد وارد بخش ساخت مدل و کامل شدن چرخه ی MVC خواهیم شد. خیلی زود متوجه می شوید که لاراول چقدر نسبت به PHP کارها را آسان تر کرده است. اما قبل از هر چیز دانستن برخی مقدمات ضروری هستند.

تنظیمات اولیه

از فایلی به نام **env.** در ریشه ی پروژه شروع می کنیم. در این فایل تنظیمات لازم برای اتصال پروژه به بانک اطلاعاتی را درج می کنند. فرض کنید پارامتر زیر را در آن که نام دیتابیس هست تغییر دادیم:

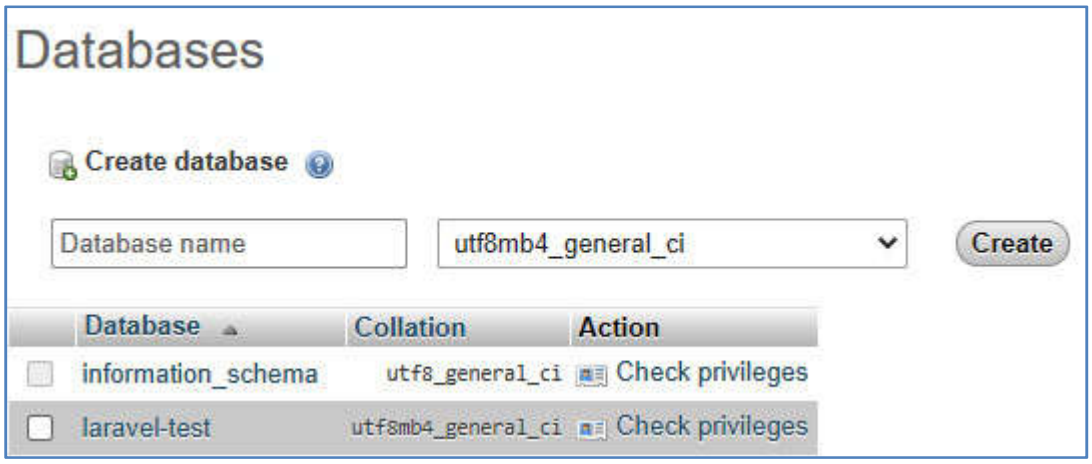
DB_DATABASE=laravel-test

در این صورت لازم است، در صورت بودن در حالت اجرای سِرور، با CTRL+C خارج شده و دوباره سِرور مخصوص لاراوال را با دستور زیر راه اندازی کنیم:

php artisan serve

تا فایل env. دوباره خوانده شود. در مرحله ی بعد کافیست فقط به پنل مخصوص دیتابیس مثلاً در phpMyAdmin که با زمپ نصب کرده اید رفته و یک بانک اطلاعاتی به همین نام بسازید.

دقت کنید که UTF8 جزو تنظیمات دیتابیس منظور شده باشد تا امکان ذخیره و نمایش کاراکترهای فارسی فراهم باشد.

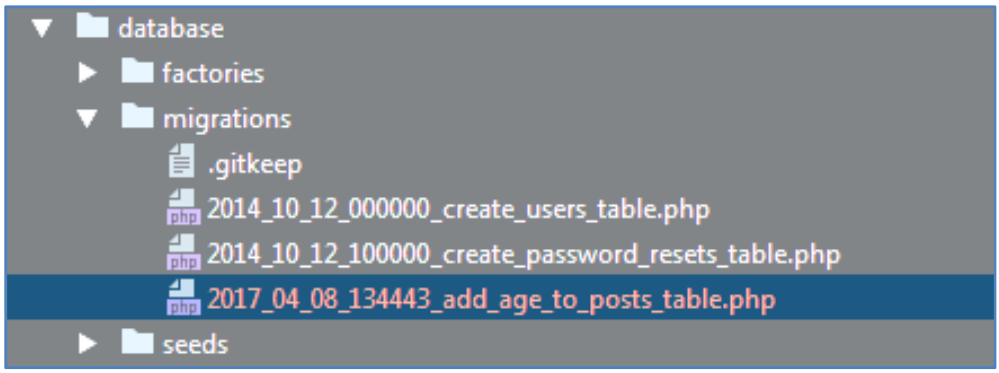


بعد از ساختن بانک اطلاعاتی laravel-test حتماً به صورت بالا در لیست دیتابیس ها باید دیده شود. از این به بعد دیگر با پنل بانک اطلاعاتی کاری نخواهید داشت. در واقع ساخت جداول و فیلدهای لازم را به صورت خودکار خود لاراول انجام خواهد داد!

آشنایی با مایگریشن

باز در بخش اسناد سایت لاراوال در قسمت Database وارد مبحث Migrations بشوید. در اینجا توضیحات لازم مفصل قید شده است.

به پروژه نگاه کنید. مسیری به نام database/migrations وجود دارد که مجموع مایگریشن های ما در آنجا قرار میگیرند. در واقع مایگریشن ها فایلهایی هستند که دستورات ساخت جداول و فیلدها در آنها بر اساس مدل های تعریف شده در پروژه قید می شوند. هر تغییری در ساختار بانک اطلاعات پروژه، مستلزم درج در یک مایگریشن (مهاجرت) جدید است. ما این بخش را به همان صورت مایگریشن نام می بریم چون اسم خاص برای یک بخش از برنامه بوده و ترجمه ی آن به فارسی کار صحیحی نیست.



اگر به نام فایلهای مایگریشن توجه کنید، در ابتدای آنها یک تاریخ و سپس شرح کاری که آن فایل انجام میدهد نوشته شده است. به عنوان مثال اگر اولین فایل نمونه مثل create_users_table را در ویرایشگر باز کنیم، متوجه می شویم که یک کلاس به همین نام درون آن ساخته شده که از کلاس Migration در کل پروژه extends شده و ارث بری دارد:

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

class CreateUsersTable extends Migration
{

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

    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

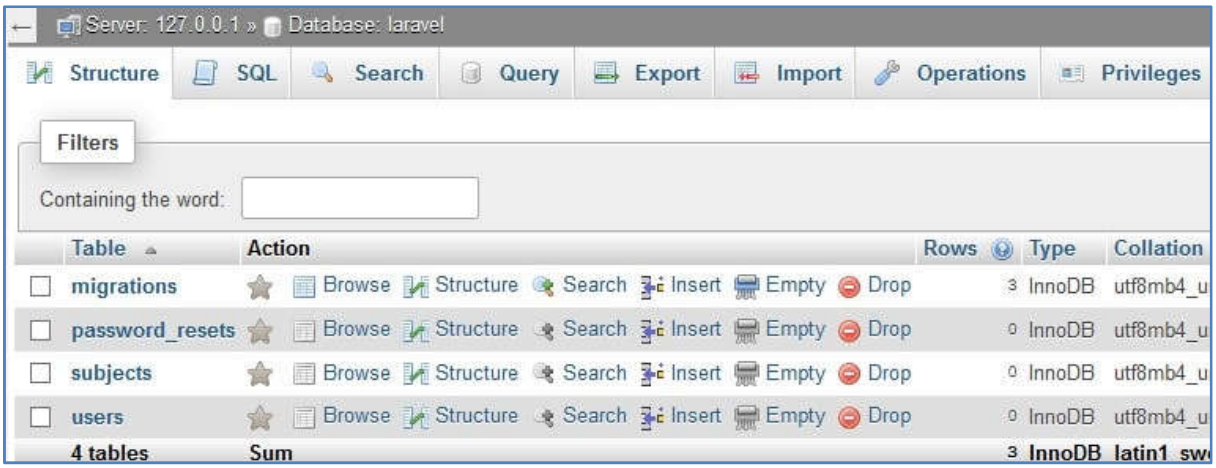
دو عملگر به نامهای up و down هم در بدنه ای این کلاس تعریف شده اند. در up فیلدها و نوع آنها با برای ایجاد ستون های جدول مشخص شده اند.

در بخش اسناد لاراول در سایت به دنبال Available column types بگردید تا شیوه ی تعریف انواع فیلدهای ممکن برای لاراول را یاد بگیرید. البته خواهید دید که نیازی به نوشتن باقی کدها توسط شما باز نخواهد بود! با خلاصه ای که برای آشنایی با ماهیت مایگریشن گفته شده، حالا به سراغ مراحل ساخت آن می رویم.

فعلاً برای اینکه بدانیم مایگریشن های موجود چگونه عمل خواهند کرد، دستور زیر را در خط فرمان و مسیر پروژه اجرا کنید:

php artisan migrate

بعد از اجرای این دستور اگر به سراغ بانک اطلاعاتی خود بروید متوجه خواهید شد که جداولی در آن به وجود آمده اند:



حالا با اجرای مجدد این دستور با پیام زیر مواجه خواهید شد:

Nothing to migrate.

یعنی ایجاد جداول یک بار اتفاق خواهد افتاد. اگر به هر دلیلی تغییری در ساختار مایگریشن ها حاصل شد و نیاز به اجرای مجدد داشتیم چطور؟ بر فرض در میانه ی راه به خطایی برخورد کرده ایم و کار مایگریشن ها ناقص انجام شده. در اینصورت دستور زیر را اجرا کنید:

php artisan migrate:fresh

اخطار: توجه داشته باشید با اجرای دستور فوق، تمام جداول و اطلاعات آنها پاک شده و دوباره ساخته می شوند. بنابراین اگر اطلاعاتی در آنها وارد کرده اید مراقب استفاده از این دستور باشید.

مهمترین فایده ی حضور این مایگریشن ها تغییراتی است که در ساختار پروژه انجام می دهیم. به این ترتیب هم خود ما و هم دیگرانی که روند کار را رصد می کنند، آگاه خواهند بود که چه تغییراتی به ترتیب تاریخ روی پروژه رخ داده است. اما فایده ی مهمتر همانطور که روشن است، بی نیاز شدن ما از تهیه ی نسخه ی پشتیبان برای دیتابیس است! در واقع پروژه را هر کجا که بخواهیم نصب کنیم، کافیست یک بار مایگریشن های آن را اجرا کنیم تا ساختار مورد نظر ما در بانک اطلاعات ایجاد شود.

ساخت مایگریشن

برای شروع فرض کنید می خواهیم جدولی به نام posts ایجاد کنیم:

php artisan make:migration create_posts_table

با اجرای دستور فوق یک مایگریشن به پوشه ی migration اضافه می شود به تاریخ امروز و نامی که در بالا بخش آخر دستور داده ایم. فایل را در ویرایشگر باز کنید. هر دو متد up و down را درون آن خواهید دید. در بخش عملگر up که باید فیلدهای جدول را تعریف کنیم و در بخش down هم دستوری نوشته شده که در هنگام حذف این جدول و خنثا کردن این مایگریشن کاربرد دارد.

دقت کنید که درون up چنین چیزی را داشته باشید:

```
Schema::create('posts', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->timestamps();
});
```

همینطور که روشن است به صورت پیشفرض فقط دو پارامتر تعریف شده است. یکی id و دیگری بخشی به نام timestamps که توسط آن در اصل دو فیلد به جدول شما اضافه می شود. یکی برای درج تاریخ ساخت یک رکورد و دیگری آخرین تغییر در آن. این دو فیلد در انتهای فیلدهای جداول قبلی مثل users قابل مشاهده هستند که به نام created_at و updated_at دیده می شوند.

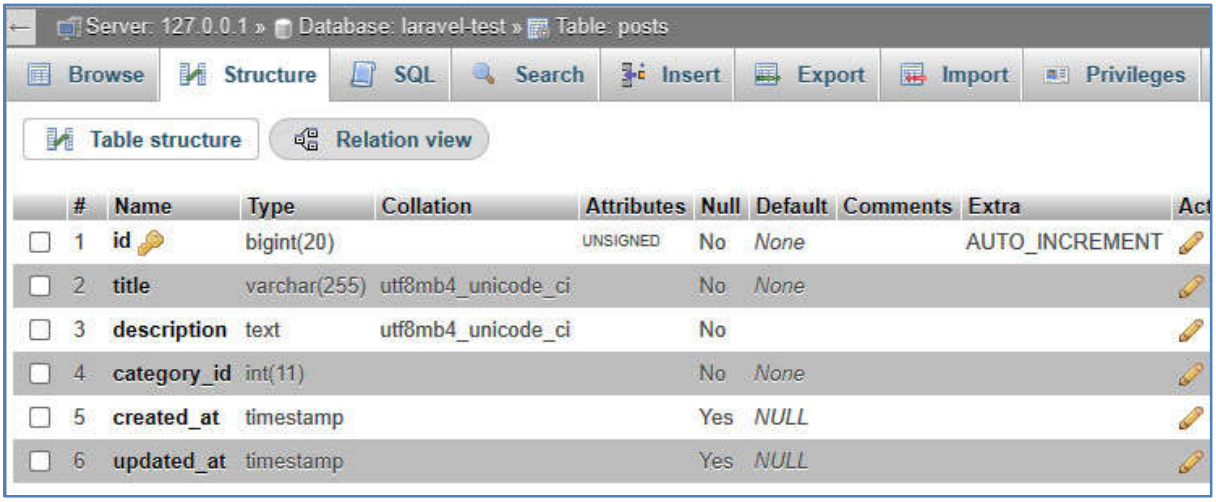
حالا فیلدهای جدید را اضافه می کنیم:

```
Schema::create('posts', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('title');
    $table->text('description');
    $table->integer('category_id');
    $table->timestamps();
});
```

فعلاً همین فیلدها کافیست. با توجه به اینکه مایگریشن جدید ساخته شده آن را اجرا می کنیم:

php artisan migrate

به صورت خودکار علاوه بر جدول posts فیلدهای زیر در ساختار این جدول ایجاد می شوند:



اما گاهی لازم است که تغییرات را به اصطلاح عقبگرد[۲۹] کنیم. کافیست برای فقط یک مرحله عقبگرد، دستور زیر را اجرا کنیم:

php artisan migrate:rollback

[29] Undo

و با اجرای دستور فوق پیامی مشابه پیام زیر دریافت می کنید:

Rolling back: 2020_07_14_093607_create_posts_table

Rolled back: 2020_07_14_093607_create_posts_table (0.1 seconds)

که یعنی یک مرحله ی قبل را خنثا و برگرداندم. اگر بنا باشد بیش از یک مرحله مثلاً ۳ مرحله به عقب برگردیم فرمت دستور به صورت زیر خواهد بود:

php artisan migrate:rollback --step=3

و حالا دوباره جدول post را با دستور زیر برگردانید:

php artisan migrate

حالا فرض کنید می خواهیم یک فیلد اضافه کنیم. ممکن چون در جداول دیتا وارد کرده ایم هم امکان rollback و نظایر آن نباشد. بنابراین یک مایگریشن جدید به صورت زیر ایجاد می کنیم:

php artisan make:migration add_user_id_to_posts

محتوای این مایگریشن جدید را با قبلی که ساخته بودیم و اول اسمش به جای add کلمه ی create داشت مقایسه کنید:

```
Schema::table('posts', function (Blueprint $table) {
    //
});
```

همانطور که می بینید داخل متد up این بار خبری از ساختن جدول جدید نیست. داخل این بخش مانند قبل با دستور $table کافیست فیلد مورد نظر را تعریف کنیم:

```
$table->integer('user_id')->after('id');
```

توجه کنید اگر بخش after('id') را وارد نکنیم، فیلد جدید در پایان جدول و انتهای آن بعد از فیلدهای موجود اضافه خواهد شد. ولی با این دستور درست بعد از فیلد id قرار خواهد گرفت.
حالا وقت آن رسیده که بعد از ذخیره ی مایگریشن باز آن را اجرا کنیم:

```
php artisan migrate
```

و اگر باز به جدول مراجعه کنیم خواهیم دید که فیلد جدید اضافه شده است.
انواع مایگریشن ها به تعداد انواع تغییرات در دیتابیس می توانند تعریف شوند. نمونه های دیگر را باز می توانید از بخش اسناد سایت ببینید.
اما نکته ی دارای اهمیت این است که مایگریشن به هیچ وجه بخش مدل از MVC محسوب نمی شود. چرا که فقط یک بار در راه اندازی اولیه ی پروژه کاربرد دارد.

خواندن از بانک

لاراول همانطور که فهمیده اید از الگو یا معماری MVC برای کار با دیتا بهره می برد. اما در عین حال دست برنامه نویس را باز می گذارد تا هر طور که صلاح می داند این بخش ها را به کار بگیرد. به عنوان مثال در بخش مسیردهی دیدیم که شما را مجبور نمی کنید برای اتصال یک مسیر به یک ویو، حتماً کنترلر داشته باشید و می توانید مستقیم این اتصال را برقرار کنید. در مورد کار با بانک اطلاعاتی هم همینطور است. پس ابتدا روش بدون مدل را بررسی می کنیم.
در بخش اسناد لاراول باز در بخش Database دو روش معرفی شده اند. یکی Query builder و دیگری Eloquent ORM معرفی شده است. روش دوم همان استفاده از مدل است که بعد از یادگیری بخش اول به آن پرداخته می شود. اما نکته ی مهم در مورد بخش کوئری آن است که بیشتر دستورات و تمرین های آن را می توان در بخش مدل هم به کار برد. بنابراین قبل از شروع مدل حتماً این بخش را مطالعه کنید تا روشهای واکشی اطلاعات به ترتیب و شکل های مختلف را از بانک یاد بگیرید.

مقدمات ارتباط با بانک

اگر به نمونه کدی که در بخش Query builder در اسناد معرفی شده نگاه کنیم متوجه خواهیم شد که اولاً این دستورات درون یک فایل کنترلر نوشته می شوند. و دوم اینکه در کدها یک کلاس با نام DB فراخوانی[۳۰] می شود.
بعد از این است که با یک یا دو خط کد می توان سطرهای[۳۱] اطلاعاتی مورد نظر خود را انتخاب[۳۲] کرده و بدون نوشتن دستورات اضافه ی دیگری که در PHP برای بازکردن[۳۳] دیتابیس، بستن[۳۴] یا

[30] use
[31] Records
[32] Select
[33] Open
[34] Close

واکشی[۳۵] اطلاعات وارد می کردیم، درون یک متغیر قرار داده و به سمت ویو فرستاد. همه ی آن دستورات قدیمی در واقع در کلاس هایی که داخل کدهای لاراول نوشته شده قرار دارند و به صورت خودکار استفاده می شوند.
خوب پس با ساخت کنترلر مانند آنچه پیش از این تمرین کرده ایم، شروع می کنیم:

php artisan make:controller PostController

و به این ترتیب یک کنترلر در مسیر Http/Controllers ساخته می شود. توجه کردید که کنترلرها به صورت استاندارد با نام مفرد ساخته می شوند و مثل نام جدول دیتابیس که Posts بود، آن را جمع انتخاب نکردیم. در واقع در نام کنترلر بخش اول یک نام مفرد و در بخش دوم خود عبارت کنترلر قرار داده می شود.
به سراغ فایل web.php در مسیر Routes هم بروید:

```
Route::get('/posts', 'PostController@index');
```

و همانطور که اکنون می دانید خط بالا به دنبال عملگر یا متد index درون کنترلر می گردد. پس درون فایل کنترلر این متد را می نویسیم:

```
class PostController extends Controller
{
   public function index() {
       $pageTitle = "پست ها";
       return view('posts', compact('pageTitle'));
   }
}
```

خوب همانطور که روشن است متد ایندکس در حال پاس کردن یک متغیر به نام pageTitle به یک ویو به نام posts هست. پس فایل ویوی مربوط یعنی posts.blade.php را هم در مسیر resources/views باید بسازیم:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title></title>
</head>
<body>
```

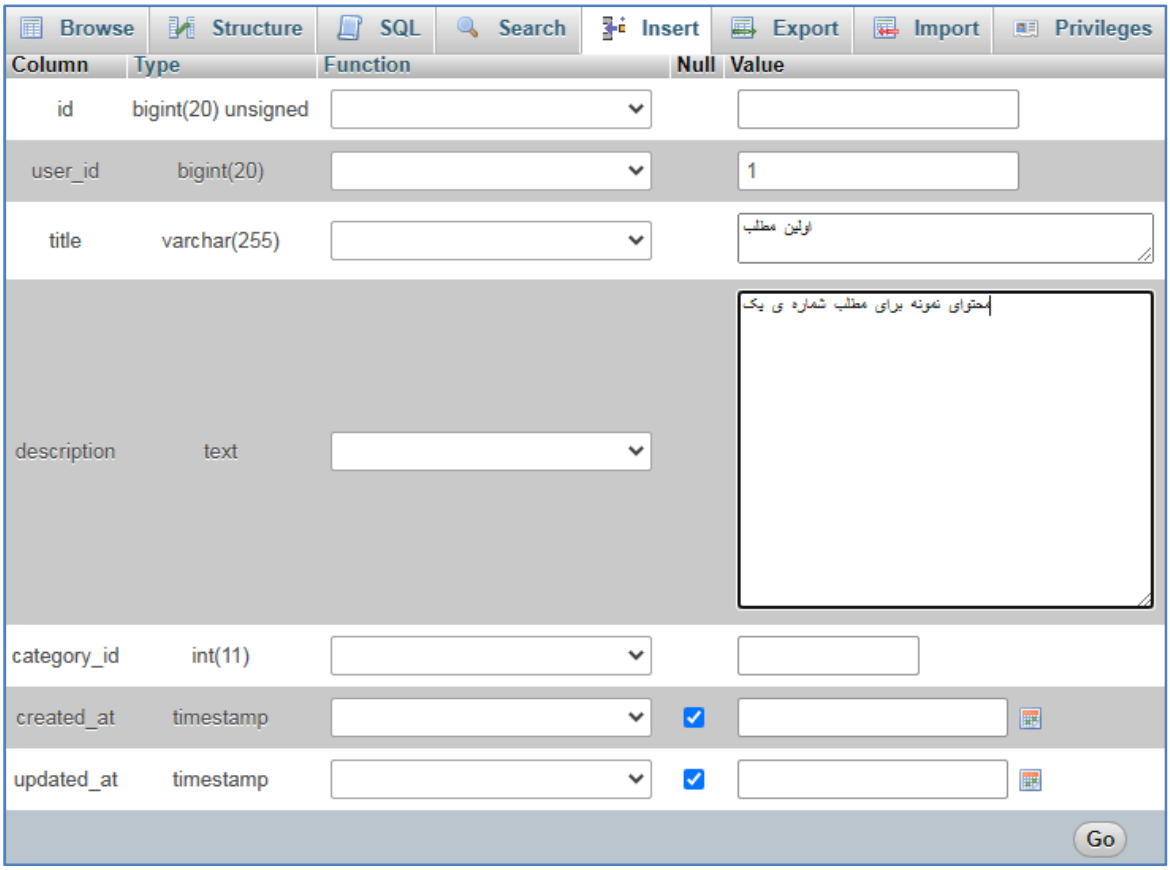
[35] Fetch

```
    {{$pageTitle}}
</body>
</html>
```

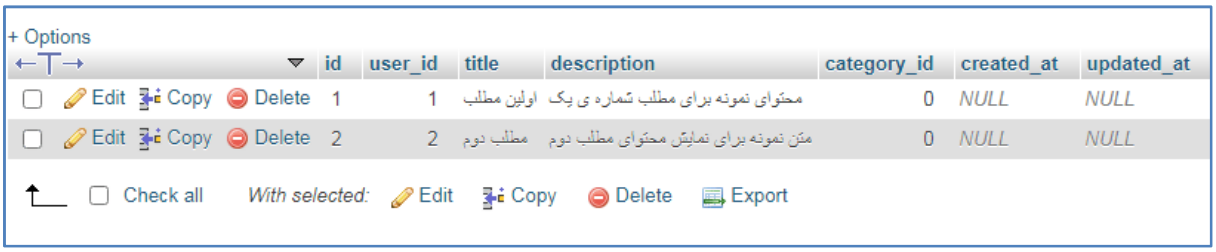
تا به اینجا همه چیز تمرین مطالب گذشته ی ما بوده است. یک بار برای امتحان، مسیر زیر را اجرا می کنیم:

Localhost:8000/posts

اگر بدون خطا اجرا شد، آماده می شویم تا بخش دیتابیس را هم در آن پیاده سازی کنیم.
به سراغ دیتابیس در phpMyAdmin بروید و بعد از انتخاب جدول posts در دیتابیس مرتبط با پروژه، از منوی بالا، روی Insert کلیک کنید:



این سه فیلد را برای نمونه پر کنید. می توانید چند نمونه ی دیگر به این شکل مطلب نمونه به دیتابیس بدهید تا برای خواندن و نمایش روی صفحه استفاده کنیم. در پایان با زدن کلید Go در پایین مطالب در جدول ذخیره و قابل مشاهده خواهند بود:



+ Options

	id	user_id	title	description	category_id	created_at	updated_at
Edit Copy Delete	1	1	اولین مطلب	محتوای نمونه برای مطلب شماره ی یک	0	NULL	NULL
Edit Copy Delete	2	2	مطلب دوم	متن نمونه برای نمایش محتوای مطلب دوم	0	NULL	NULL

Check all With selected: Edit Copy Delete Export

حالا همانطور که در توضیحات اولیه گفته شد برای استفاده از دستورات مخصوص دیتابیس در لاراول کلاس مربوطه را به ابتدای فایل کنترلر خودمان PostController.php اضافه می کنیم:

```
use Illuminate\Support\Facades\DB;
```

و بعد برای اینکه دستورات دیتابیس را به کار ببریم یک متغیر هم در متد **index** استفاده می کنیم:

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

در اینجا با کلاس DB جدول posts هدفگیری شده و با متد get همه ی رکوردها فراخوانده می شود. حالا توجه کنید که کل کدهای کنترلر ما باید به صورت زیر دیده شوند:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class PostController extends Controller
{
   public function index() {
       $posts = DB::table('posts')->get();
       $pageTitle = "پست ها";
       return view('posts', compact('pageTitle','posts'));
   }
}
```

و همانطور که می بینید این بار متغیر posts هم به ویوی مربوطه ارسال شده است.
اگر در فایل ویو داشته باشیم:

```
{{$posts}}
```

تمام محتویات آرایه ی posts بدون هیچ آرایشی و با فرمت زیر به نمایش در می آید:

[{"id":1,"user_id":1,"title":"\u0627\u0648\u0644\u06cc\u0646 \u0645\u0637\u0644\u0628
","description":"\u0645\u062d\u062a\u0648\u0627\u06cc \u0646\u0645\u0648\u0646\u0647
\u0628\u0631\u0627\u06cc \u0645\u0637\u0644\u0628 \u0634\u0645\u0627\u0631\u0647 پست ها

\u06cc
\u06cc\u06a9","category_id":0,"created_at":null,"updated_at":null},{"id":2,"user_id":2,"title":"\u06
45\u0637\u0644\u0628 \u062f\u0648\u0645","description":"\u0645\u062a\u0646
\u0646\u0645\u0648\u0646\u0647 \u0628\u0631\u0627\u06cc
\u0646\u0645\u0627\u06cc\u0634 \u0645\u062d\u062a\u0648\u0627\u06cc
\u0645\u0637\u0644\u0628
\u062f\u0648\u0645","category_id":0,"created_at":null,"updated_at":null}]

طبیعی است که ما تک تک داده ها را باید بتوانیم از درون آرایه واکشی کنیم تا به صورت دلخواه و مرتب به نمایش در آوریم. برای این منظور به سراغ یک حلقه می رویم و نمایش قبلی را در ویو به صورت زیر تغییر می دهیم:

```
<body>
    {{$pageTitle}}<br><hr>
    @php
        foreach ($posts as $post) {
           echo $post->title.'<br>';
           echo $post->description.'<hr>';
        }
    @endphp
</body>
```

خوب اول اینکه به جای php?> از @php و در انتها هم از @endphp استفاده شده که دستورات لاراولی هستند. پس از آن یک حلقه ی foreach از بین هر عضو آرایه ی posts یکی یکی درون متغیر post ریخته و title و description را از هر کدام جدا کرده و نمایش میدهد. بین هر پست هم با تگ hr یک خط رسم شده است.

به این ترتیب ارتباط ما برای نمایش اطلاعات از جدول درون دیتابیس به خوبی برقرار شده است. حتا می توانیم بجای حلقه ای که به زبان PHP نوشته شده، با فرمت لاراول هم بنویسیم:

```
@foreach ($posts as $post)
   {{$post->title}}<br>
   {{$post->description}}<hr>
@endforeach
```

بسیار ساده و جمع و جورتر!

انواع کوئری

حالا در واقع باید دستوراتی را هم که به صورت سفارشی در بانکهای اطلاعاتی رابطه ای[۳۶] داده ها را با قالب خاص به ما بر می گردانند، تجربه کنیم. در اسناد سایت لاراول از بخش Chunking Results به بعد در قسمت Query builder نمونه های متعددی را در ادامه ی توضیحات خواهید دید.

[36] Relational databases

از دستور count برای شمارش رکوردها استفاده می کنیم. بخشی را به متد ایندکس اضافه می کنیم:

```
public function index() {
    $posts = DB::table('posts')->get();
    $postsCount = DB::table('posts')->count();
    $pageTitle = "ها پست";
    return view('posts', compact('pageTitle','posts','postsCount'));
}
```

و حالا به ویو هم بخشی را برای نمایش تعداد پست ها اضافه می کنیم:

```
<body>
    {{$pageTitle}}<br>
    {{$postsCount}}<br>
    <hr>

        @foreach ($posts as $post)
            {{$post->title}}<br>
            {{$post->description}}<hr>
        @endforeach
</body>
```

نتیجه نمایش تعداد پست ها در بالای صفحه است. در جریان هستید که انجام این کارها با PHP به همین سادگی نیستند.

حالا همانطور که متوجه شده اید ترتیب نمایش پست ها از اولین به آخرین است. ممکن است ما برای نمایش در یک وبلاگ نیاز داشته باشیم که آخرین پست را اول از همه و بعد از آن به ترتیب پست قبلی را قرار دهیم که قدیمی تر است. برای این منظور نیاز به یک مرتب سازی با دستور orderBy و تغییر کوچکی در خواندن پست ها داریم:

```
$posts = DB::table('posts')->orderBy('id','DESC')->get();
```

همانطور که ملاحظه خواهید کرد پس از اجرا، آخرین پست یعنی پست دوم را اول نمایش میدهد. چون ترتیب نمایش به صورت پیشفرض صعودی[۳۷] است و ما آن را نزولی[۳۸] کرده ایم.

ممکن است در یک پروژه خواسته باشیم که رکوردی برگردانده شود که بیشترین مقدار در یک فیلد آن ثبت شده باشد. مثل یک فاکتور فروش که گران ترین کالا را بخواهیم از آن استخراج کنیم. در این صورت از دستور max استفاده می کنیم. مثلاً برای مثال قبلی اگر روی فیلد id بیشترین عدد را بخواهیم بدانیم کافیست خط زیر را به کنترلر اضافه کنیم:

[37] Ascending
[38] Descending

```
    public function index() {
        $maxId = DB::table('posts')->max('id');
        $posts = DB::table('posts')->orderBy('id','DESC')->get();
        $postsCount = DB::table('posts')->count();
        $pageTitle = "ها پست";
        return view('posts', compact('pageTitle','posts','postsCount','maxId'))
;
    }
```

در این صورت اگر خط زیر را هم به بخش ویو اضافه کنیم:

```
{{$maxId}} :Big id
```

بیشترین شماره ی id را نمایش میدهد.

اگر پیش از این در PHP با دستور SELECT در فراخوانی اطلاعات از دیتابیس کار کرده باشید می دانید که بخشی از این دستور به نام WHERE برای انتخاب رکوردهایی با شرط خاص به کار می رود. در اینجا یعنی لاراول نیز چنین امکانی وجود دارد. کافیست در کنترلر به جای فراخوانی همه ی رکوردها، قبل از get به صورت زیر از where استفاده شود:

```
// $posts = DB::table('posts')->orderBy('id','DESC')->get();
 $posts = DB::table('posts')->where('id',2)->get();
```

در اینجا خط قبلی که همه ی رکوردها را فراخوانی میکرد به صورت کامنت در می آوریم تا عمل نکند و خط جدید فقط پستی را که در آن id برابر ۲ باشد می خواند.
البته در واقع دستور where دارای سه بخش ورودی است. بنابراین اگر کد را به صورت زیر تغییر دهید:

```
->where('id','<',3)
```

تمام رکوردهایی که در آن مقدار id کوچکتر از ۳ باشد نمایش داده می شوند.
بخشی را که به صورت کامنت در آورده اید با این کوئری جدید جابجا کنید.

حالا به یکی از بهترین دستورات این بخش یعنی **paginate** می رسیم که برای صفحه بندی نمایش رکوردها در زمانی که تعداد آنها بالا می رود است. ما با همین دو رکورد هم می توانیم این بخش را تست کنیم. همانطور که همیشه در این مجموعه خاطرنشان شده، هر دستور لاراول کار دهها خط کد در PHP را انجام می دهد و این بهترین نمونه از این دستورات است که به جای get به کار گرفته می شود:

```
$posts = DB::table('posts')->orderBy('id','DESC')->paginate(1);
```

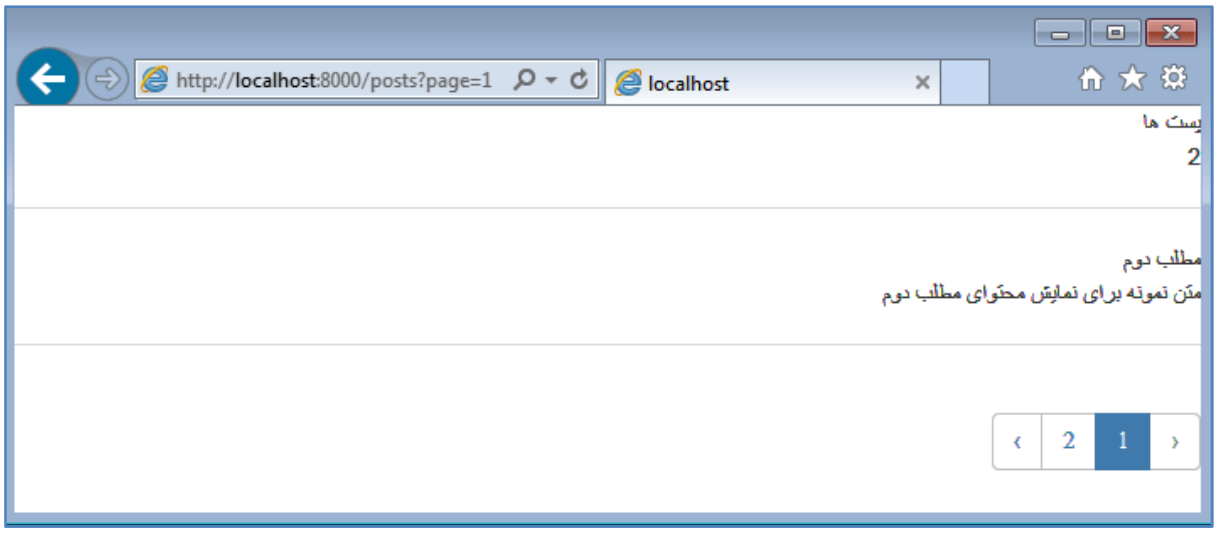
عدد داخل پرانتز تعداد هر پست در صفحه را نشان می دهد. تنها کار باقیمانده آن است که در بخش ویو هم قبل از بستن تگ body نمایش منوی صفحه بندی را داشته باشیم:

```
{{$posts->links()}}
```

با اجرای برنامه، دکمه های صفحه بندی[۳۹] هم به پایین صفحه اضافه می شوند. که البته چون استایل و آرایش خاصی به آنها نسبت داده نشده به صورت عمودی دیده می شوند، ولی درست کار می کنند. برای اصلاح نمایش هم ابتدا خط زیر را داخل تگ head در ویوی خود اضافه کنید:

```
<link rel="stylesheet" href="bootstrap.css">
```

بعد فایل **bootstrap.css** را که مربوط به فریمورک معروف بوت استرپ است از اینترنت دانلود کرده و در پوشه ی public کپی کنید. حتا می توانید نسخه ی راستچین را که برای فارسی و عربی مناسب است به کار ببرید و هنگام سرچ در گوگل عبارت rtl یعنی راست به چپ را هم اضافه کنید. در اینصورت نتیجه باید به شکل زیر باشد:



تا اینجا ساده ترین روش برای خواندن و نمایش اطلاعات از بانک اطلاعاتی را دیده اید. نمایشی که تا حد امکان کامل و بی نقص است. حالا آماده می شویم که با روش اصولی تر طبق معماری MVC یعنی کار با دیتا از طریق ایجاد مدل آشنا بشویم.

ساخت مدل

باز به بخش اسناد سایت لاراول و این بار Eloquent ORM می رویم. حالا فرض کنید جدولی به نام orders یا سفارشها قرار است داشته باشیم. طبعاً برای داشتن یک چرخه ی کامل کاری

[39] Pagination

در MVC باید هر سه بخش ویو، کنترلر و مدل را بسازیم. روشن است برای ساختن کنترلر این جدول باید دستور زیر را اول اجرا کنیم:

php artisan make:controller OrderController

بعد از اجرای این دستور همانطور که حالا دیگر می دانید در مسیر app/Http/Controllers فایلی ایجاد می شود. بعد از این نوبت به ساخت مایگریشن برای ایجاد جدول می رسد. طبعاً شما می دانید که این مایگریشن هم چطور باید ساخته شود. بعد از آن در مسیر database/migrations هم فایل مورد نظر ما ساخته خواهد شد. اما با توجه به روشی که از این به بعد قرار است پی بگیریم، دستور ساخت مایگریشن را به صورتی متفاوت اجرا خواهیم کرد. پس خط زیر را اجرا نکنید فقط به یاد بیاورید که پیش از این مایگریشن را به صورت زیر برای چنین جدولی می نوشتیم:

php artisan make:migration create_orders_table

اگر هم ساخته اید، به راحتی از پوشه ی مربوطه در پروژه آن را حذف کنید. حالا اما به صورت زیر می نویسیم:

php artisan make:model Order -m

در اینجا مدل را هم مانند کنترلر به صورت فرد نامگذاری می کنیم یعنی Order نه Orders. اما سوئیچ m- اتفاق جالبی را رقم زده است. کافی است به پوشه ی مایگریشن ها توجه کنید تا متوجه شوید که مایگریشن مورد نظر برای این مدل هم به صورت خودکار تولید شده است! اما خود فایل مدل همان Order.php است که مانند مدل های دیگر در ریشه ی پوشه ی app ساخته شده است. این مسیرها قابل تغییر هستند که جلوتر به آن خواهیم پرداخت.

همانطور که متوجه شده اید با ساختن مایگریشن به صورت خودکار، در واقع لاراول به دنبال آن است که مدل Order را به جدولی به نام orders در بانک اطلاعات متصل کند. اما اگر نخواهیم چنین باشد چه؟ دستوراتی برای تغییر این پیشفرض ها وجود دارند. در همین مثال اگر مدل را باز کنیم، کافیست داخل آن:

```
class Order extends Model
{
    //
}
```

به جای // مثلاً بنویسیم:

```
protected $table = 'invoices';
```

این خط یعنی که جدول مرتبط با این مدل را بجای orders با نام invoices در نظر بگیر. یعنی اگر چنین خطی نوشته نشود به صورت خودکار جدول مرتبط با مدل Order می شود orders.
یا در مورد id و فیلد خودکار کد کامپیوتری که معمولاً در ابتدای فیلدهای جداول ساخته می شوند. به صورت پیشفرض این فیلد هم با نام id در نظر گرفته می شود. فرض کنید در مثال فعلی هم این فیلد را با نام order_id ساخته باشیم. یعنی در مایگریشن به جای id نام order_id را به آن داده ایم. باید بدانیم که در این صورت مدل Order برای کار با چنین جدولی به مشکل بر خواهد خورد چون به صورت پیشفرض به دنبال id می گردد. راه حل باز وارد کردن دستور زیر در مدل است:

```
protected $primaryKey = 'order_id';
```

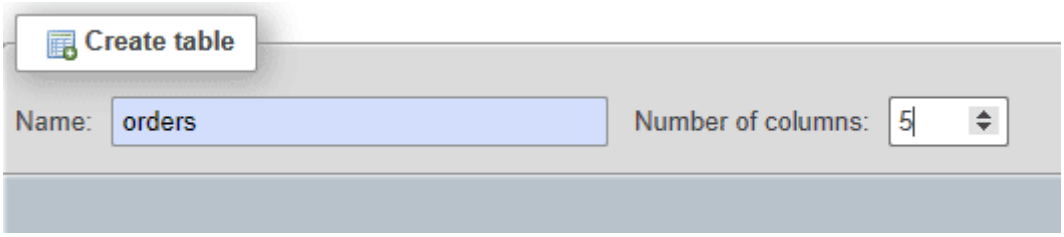
و همانطور که می بینید با همین دستورات یک خطی همه ی تنظیمات به راحتی انجام می شوند.

اگر خاطرتان باشد در ساخت مایگریشن بخشی به نام timestamps وجود داشت که دو فیلد تاریخ در انتهای جداول ایجاد می کرد. یکی برای تاریخ ساخت جدول و دیگری تاریخ آخرین به روزرسانی. ممکن است ما جدولی طراحی کنیم که در آن نخواهیم از این ویژگی بهره ببریم. در همین مثال فعلی اگر این بخش را در جدول نسازیم، باز مدلی که به صورت پیشفرض برای کار با آن ساخته ایم، برای ارتباط به مشکل بر خواهد خورد چون مدل ما دنبال این فیلدها می گردد. برای رفع این مشکل هم کافیست که دستور زیر را باز در مدل وارد کنیم:

```
protected $timestamps = false;
```

که باعث می شود بررسی فیلدهای یاد شده، خاموش باشد. روشن است که اگر این کار را نکنیم، مدل به صورت خودکار هنگام ورود اطلاعات در جدول به دنبال این دو فیلد می گردد و اگر آنها را پیدا نکند خطا می دهد. اگر هم آنها را بیابد، باز به صورت خودکار پُر می کند.

اما به سراغ تست مدل خود برویم. برای اینکه از سرعت بیشتری برخوردار باشیم، بدون بهره گیری از مایگریشن و به صورت دستی جدول orders را با ۵ فیلد به صورت زیر در دیتابیس خیلی سریع بسازید:



بعد باید ساختاری مانند زیر بسازید:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
☐	1	id	int(11)			No	None		AUTO_INCREMENT
☐	2	user_id	int(11)			No	None		
☐	3	title	varchar(256)	utf8mb4_general_ci		No	None		
☐	4	amount	int(11)			No	None		
☐	5	order_id	int(11)			No	None		

و به سراغ Insert رفته و دیتاهای آزمایشی زیر را هم وارد کنید:

+ Options

	id	user_id	title	amount	order_id
☐ Edit Copy Delete	1	1	لپ تاپ ایسوس مدل فلان	25000000	450
☐ Edit Copy Delete	2	2	مانیتور سامسونگ	410000	460

حالا که دو سفارش نمونه ثبت کردیم،
به سراغ مسیرها در فایل web.php بروید:

```
Route::get('/orders','OrderController@index');
```

اگر در مرورگر نشانی /orders بعد از ریشه صدا شود، متد index از OrderController قرار است اجرا شود. پس باید این متد در کنترلر مربوطه ساخته شود:

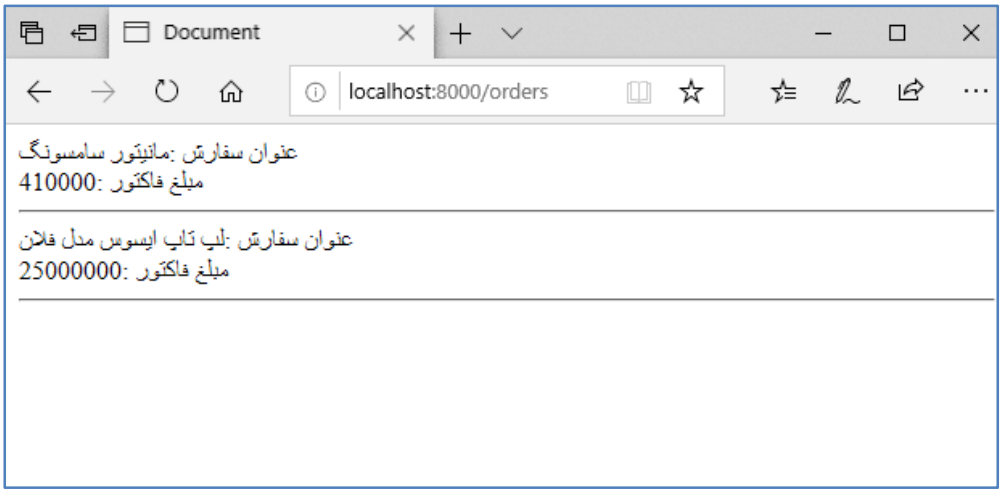
```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class OrderController extends Controller
{
    public function index(){
        $pageTitle = "صفحه سفارش ها";
        $orders = DB::table('orders')->orderBy('id','DESC')->get();
           return view('orders', compact('pageTitle','orders'));
       }
}
```

بعد ویوی مربوطه را هم باید در مسیر resources/views با نام orders.blade.php بسازیم:

```
<body>
@foreach ($orders as $order)
  عنوان سفارش :{{$order->title}}<br>
```

```
  مبلغ فاکتور :{{$order->amount}}<hr>
@endforeach
</body>
```

حالا اگر در مرورگر نشانی localhost:8000/orders را بزنیم، باید دو سفارش خود را در صفحه ببینیم:



تا اینجا هنوز با مدل کاری نکرده ایم. در واقع از همان روش Query builder بهره برده ایم. برای شروع کار با مدل لازم است که خط زیر را از کنترلر خود حذف کنیم:

```
use Illuminate\Support\Facades\DB;
```

چون دیگر با کلاس DB کاری نداریم. بلکه باید خط زیر را این بار به جای خط بالا اضافه کنیم:

```
use App\Order;
```

و از آنجا که کلاس DB هم دیگر کاربردی ندارد، در متد index باید کلاس فراخوانی جدول orders به صورت زیر باشد:

```
$orders = Order::orderBy('id','DESC')->get();
```

این بار خود کلاس Order همه کار را مدیریت می کند. و اگر صفحه را رفرش کنیم نتیجه مانند قبل خواهد بود. با این تفاوت که روش MVC به طور کامل پیاده سازی شده است. درخواست به کنترلر، از آنجا به مدل و بعد از دریافت نتیجه از مدل دوباره خروجی به ویو برگردانده می شود. با اینکه فایل مدل که آن را به صورت خودکار ساخته ایم، تقریباً خالی است و هیچ دستور خاصی در آن ننوشته ایم اما همین فایل، فرآیند کامل دسترسی به دیتابیس را به انجام رسانده است.

توجه داشته باشید از تمام دستوراتی که در بخش قبل و بدون مدل استفاده کردید مثل count در اینجا هم می توانید استفاده کنید. مثلاً برای داشتن تعداد سفارش ها در همین مثال:

```
$ordersCount = Order::count();
```

و بسیار خلاصه تر از روش قبلی می توان دریافت اطلاعات را از بانک مدیریت کرد.

یک پروژه ی نمونه

در اینجا با اطلاعاتی که تا این لحظه به دست آورده ایم شروع به ساخت یک روال کامل برای کار با جدول اطلاعاتی می کنیم. روال هایی مثل فرم ورود اطلاعات، بخش خواندن و نمایش، حذف یا و ویرایش.دستور جالبی که قرار است برای شروع کار اجرا کنیم، هر سه بخش کنترلر، مدل و مایگریشن را با هم می سازد:

php artisan make:model Category -a

علاوه بر ساخت مدلی به نام Category که قرار است دسته بندی های سایت خود را به طور مثال در آن وارد کنیم، کنترلر و مایگریشن هم ساخته می شوند! یعنی پیامهای صادره بعد از اجرای دستور فوق به شرح زیر خواهند بود:

```
Model created successfully.
Factory created successfully.
Created Migration: 2020_07_22_085342_create_categories_table
Seeder created successfully.
Controller created successfully.
```

و ممکن است از بین عملیاتی که در بالا انجام شده بخش هایی برای شما سؤال برانگیز باشند که مثلاً Factory چیست!؟ اگر به پوشه ی database/facories هم نگاه بیاندازید، فایل CategoryFatory.php را می بینید. در واقع این فایل برای ایجاد مقادیر دیتای آزمایشی به کار می رود تا شما را از ورود اولیه ی دیتا به صورت دستی یا از طریق پنل phpMyAdmin بی نیاز کند. البته این فرآیند باید بعد از اجرای مایگریشن و ساخته شدن جدول انجام پذیرد. در این مورد به همین معرفی بسنده می کنیم و ادامه ی آموزش را با مراحل دیگری پی می گیریم.

پس به سراغ فایل مایگریشن که با نام **_create_categories_table** تمام می شود، رفته و فیلدهای زیر را برای جدول categories به شرح زیر تعریف می کنیم:

```
    public function up()
    {
```

```
        Schema::create('categories', function (Blueprint $table) {
            $table->bigIncrements('id');     /* کد خودکار */
            $table->string('title');         /* عنوان */
            $table->text('description');     /* توضیحات */
            $table->tinyInteger('active');   /* فعال بودن */
            $table->timestamps();            /* تاریخ ایجاد و ویرایش */
        });
    }
```

که همانطور که از قبل می دانیم، بعد از ذخیره باید با دستور زیر مایگریشن خود را اجرا کنیم تا جدول با فیلدهای بالا ساخته شود:

php artisan migrate

جدول categories و فیلدهای آن ساخته شد. با توجه به اینکه در تمرینات قبل جداولی مثل orders را به صورت دستی ساخته اید، اگر به خطایی برخورد کردید، مایگریشن مربوط به آن را فعلاً پاک کنید.

باز به بخش Insert در این جدول رفته خیلی سریع اطلاعاتی را به صورت آزمایشی در آن وارد کنید:

و حالا نگاهی به کنترلر این جدول یعنی **CategoryController** بیاندازید. در ظاهر تفاوتی با کنترلرهای قبلی که پیش از این ساخته ایم دارد! در واقع به صورت خودکار متدهای لازم همه ساخته شده اند. و همچنین علاوه بر متد index که برای ارجاع به بخش ویو از آن استفاده می کردیم، متدهای دیگری هم برای ویرایش، حذف و غیره ساخته شده اند:

```
class CategoryController extends Controller
{
    public function index()
    {
        //
    }

    public function create()
    {
        //
    }

    public function store(Request $request)
    {
        //
```

```
    }

    public function show(Category $category)
    {
        //
    }

    public function edit(Category $category)
    {
        //
    }

    public function update(Request $request, Category $category)
    {
        //
    }

    public function destroy(Category $category)
    {
        //
    }
}
```

که البته در میان هر متد بخش هایی برای کامنت و توضیحات هر بخش نیز وجود دارند. بعد از متد index که می دانیم برای ارسال به صفحه و مسیر اولیه است، متد create برای بخش فرم ورود اطلاعات، متد store زمان ذخیره ی فرم ورود اطلاعات، متد show برای زمان نمایش اطلاعات فقط یک رکورد از جدول، edit برای زمان نمایش فرم ویرایش، update لحظه ی ذخیره ی فرم ویرایش، و destroy هم هنگام حذف فراخوانی می شوند.

فایل ویوی متناسب این بخش یعنی categories.blade.php را هم در مسیر ویژه ی آن ایجاد کنید:

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="bootstrap.min.css">
    <title></title>
</head>

<body>
    {{ $pageTitle }}<br>
    <hr>
    <a href="#" class="btn btn-primary">دسته جدید</a>
    <hr>
    <div class="container">
    <table class="table table-striped">
```

```
        <thead>
            <tr>
                <td>عنوان</td>
                <td>توضیحات</td>
                <td>فعال</td>
                <td>ویرایش</td>
                <td>حذف</td>
            </tr>
        </thead>

        <body dir="rtl">
            @foreach($categories as $category)
                <tr>
                    <td>{{ $category->title }}</td>
                    <td>{{ $category->description }}</td>
                    <td>{{ $category->active }}</td>
                    <td></td>
                    <td></td>
                </tr>
            @endforeach
        </body>
    </table>
    </div>
</body>
```

به سراغ مسیردهی در فایل web.php هم می رویم:

```
Route::get('/categories','CategoryController@index')->name('categories');
```

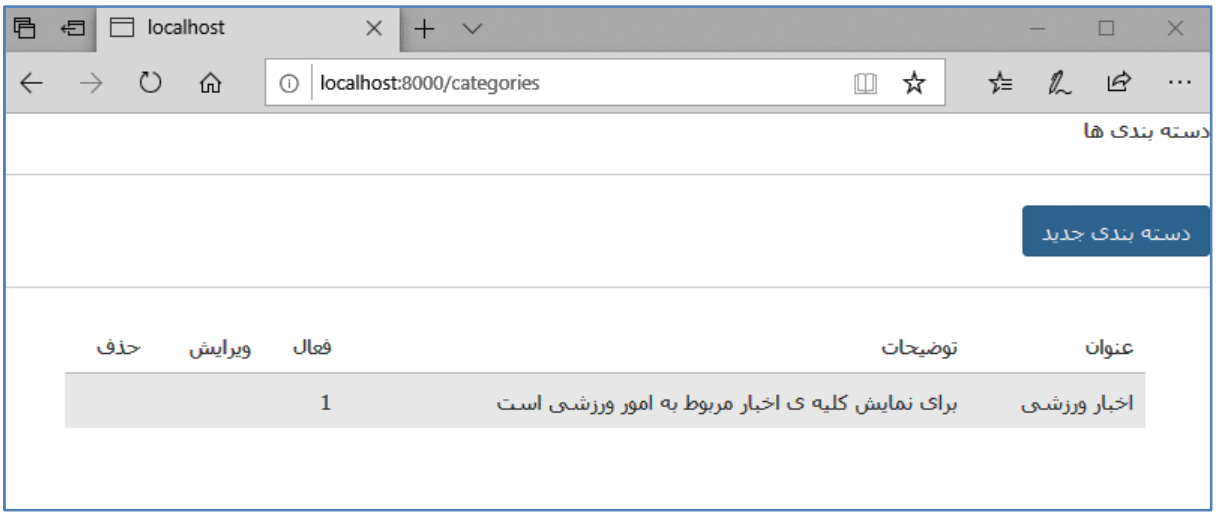
توجه کنید که بخش انتهایی یعنی name کمک می کند که بتوانیم لینک هایی برای ارتباط درست با مسیر ساخته شده در صفحه بسازیم. مثلاً در یکی وی تایپ کنیم:

```
<a href="{{route('categories')}}"> ..
```

که در این صورت به مسیر مورد نظر لینک داده می شود و اگر این بخش آخر در مسیر نباشد لاراول به ما خطا خواهد داد.
خوب حالا طبعاً باید متد ایندکس را هم در کنترلر مربوطه برای فراخوانی ویوی مناسب تغییر دهیم:

```
    public function index()
    {
        $pageTitle = "دسته بندی ها";
        $categories = Category::orderBy('id','DESC')->get();
        return view('categories', compact('pageTitle','categories'));
    }
```

خوب وقت اجراست:



و همانطور که روشن است هر سه بخش مدل، ویو و کنترلر به خوبی کار می کنند. سعی کنید که دیتای آزمایشی بیشتری وارد کنید تا نمایش بهتری در صفحه داشته باشیم.

به سراغ بخش مسیردهی رفته و مسیر جدید زیر را اضافه می کنیم:

```
Route::get('/categories/{category}','CategoryController@show')->name('show');
```

در اینجا می توانستیم به جای {category} از {id} استفاده کنیم اما این کار باعث می شود که روال های خودکار ارتباط مدل بهم بریزد. با تعریفی که در مسیر بالا کرده ایم به طور خودکار مدل خودش به دنبال کد یکتای یک رکورد می گردد. اگر جدولی به نام posts هم داشته باشیم، کافیست این بخش را به صورت {post} تعریف کنیم.

ممکن است با مثالی که در اینجا داریم یعنی دسته بندی ها، نمایش تک تک رکوردها به صورت جداگانه در صفحه ای دیگر ضروری نباشد. اما اگر این آموزش را فرا بگیرید، می توانید مثلاً برای نمایش یک جدول دیگر مثل اخبار، متن کامل هر خبر را جداگانه در صفحه ای باز کرده و نمایش دهید. بعد از تعریف این مسیر لازم است لینکی هم که صفحه ی نمایش یک رکورد را قرار است باز کند در ویو ایجاد کنیم. پس در حلقه ای که رکوردها را می خواند تغییر کوچکی ایجاد می کنیم:

```
@foreach($categories as $category)
    <tr>
        <td>
          <a href="{{route('show',$category->id)}}">
          {{ $category->title }}
          </a>
        </td>
        <td>{{ $category->description }}</td>
        <td>{{ $category->active }}</td>
```

```
                <td></td>
                <td></td>
            </tr>
        @endforeach
```

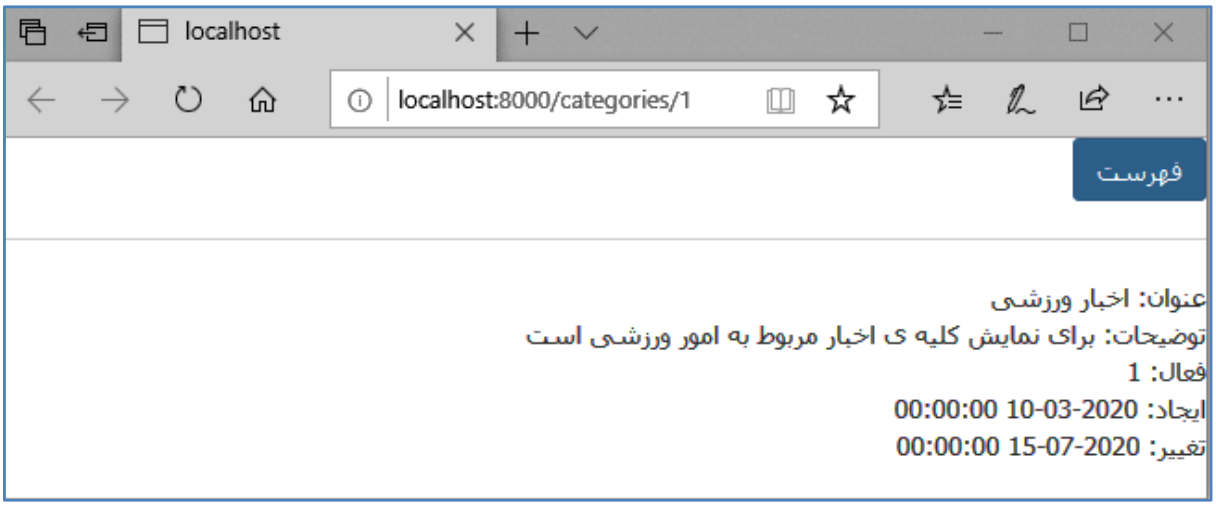
و به این ترتیب هر عنوان، به متد show برای نمایش رکوردی با id مربوط به همان رکورد لینک می شود. پس به سراغ کنترلر و متد show می رویم:

```
    public function show(Category $category)
    {
        $pageTitle = "دسته بندی ها";
        return view('category', compact('pageTitle','category'));
    }
```

همینطور باید ویوی category.blade.php را هم در مسیر ویژه ی خودش بسازیم:

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="../bootstrap.min.css">
    <title></title>
</head>

<body dir="rtl">
  <a class="btn btn-primary" href="{{route('categories')}}">
   فهرست<br>
</a>
    <hr>
عنوان: {{$category->title}}<br>
توضیحات: {{$category->description}}<br>
فعال: {{$category->active}}<br>
ایجاد: {{$category->created_at}}<br>
تغییر: {{$category->updated_at}}<br>
</body>
```

و حالا با کلیک روی عنوان هر دسته بندی، می توان صفحه ی مختص به آن را هم مشاهده کرد:



ورود اطلاعات

بی مقدمه به سراغ بخش مسیردهی رفته و مسیر جدید زیر را اضافه کنید:

```
Route::get('/categories/create','CategoryController@create')->name('create');
```

بد نیست که مسیرهای دیگر را هم یکباره تعریف کنیم:

```
Route::get('/categories/edit/{category}','CategoryController@edit')-
>name('edit');

Route::get('/categories/destroy/{category}','CategoryController@destroy')-
>name('destroy');
```

مسیرهای فوق برای ویرایش و حذف هستند. مسیر بعدی را هم برای ایجاد فرآیند ذخیره می نویسیم. این مربوط به متدی است که بعد از فراخوانی بخش create اجرا شده و عمل ذخیره در بانک را انجام خواهد داد یعنی store:

```
Route::post('/categories/store','CategoryController@store')->name('store');
```

و همانطور که می بینید به صورت post این بار تعریف شده است. برای ویرایش اما:

```
Route::put('/categories/update{category}','CategoryController@update')-
>name('update');
```

و این هم مسیری است که بعد از متد edit برای انجام مرحله ی نهایی ویرایش در بانک صدا زده می شود اطلاعات را به روزرسانی یا update می کند.

توجه کنید مجموعه ی کامل مسیرهایی که برای این جدول تعریف کرده اید (با دو مسیری هم که در بخش قبل تعریف کرده ایم، جمعاً ۷ عدد) در ایجاد نمایش و مدیریت برای جداول دیگر هم مشابه همین ها خواهند بود.

نکته: چرا بعضی مسیرها با وجود کنترلر یا ویو اجرا نمی شوند؟

اگر به چنین مشکلی برخورد کردید، کافیست بعد از چک کردن تمام دستورات و اطمینان از صحت تایپ صحیح اسامی، جای یک مسیر را در فایل web.php فقط عوض کنید! یعنی به چند خط بالاتر یا پایین تر منتقل شود. چون ممکن است به دلیل بروز خطاهای نامعلوم، یک مسیر در جای فعلی اش خوانده نشود. همینطور می توانید نهانگاه[40] برنامه را با دستور زیر پاک کنید:

php artisan config:cache

همچنین می توانید دستور php artisan serve را هم قطع و مجدد اجرا کنید.

نوبت به آن رسیده که فرم ورود اطلاعات را در ویوی مخصوص خودش بسازیم. اول در کنترلر مربوطه و در متد **create** فقط عنوان صفحه را به ویو ارسال می کنیم:

```
public function create() {
    $pageTitle = 'افزودن دسته بندی';
    return view( 'create', compact( 'pageTitle' ) );

}
```

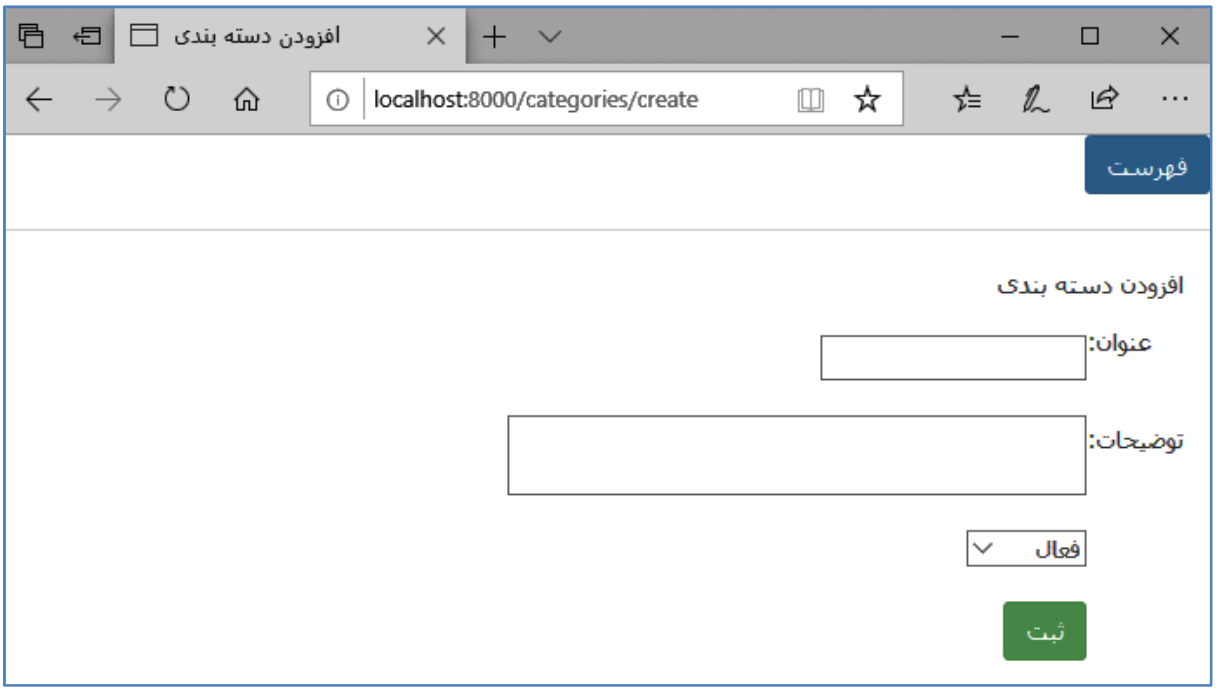
حالا ویوی create را هم در مسیر ویژه اش به نام **create.blade.php** می سازیم:

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="../bootstrap.min.css">
    <title>{{ $pageTitle }}</title>
</head>

<body dir="rtl">
    <a class="btn btn-primary" href="{{ route('categories') }}">
        فهرست<br>
    </a>
    <hr>
    <div class="container">
        {{ $pageTitle }}<br>
        <form action="{{ route('store') }}" method="POST">
            <table>
```

[40] Cache

```
                <tr>
                    <td align="left">عنوان:</td>
                    <td><br><input type="text" name="title"></td>
                </tr>
                <tr>
                    <td align="left">توضیحات:</td>
                    <td><br><textarea name="description" cols="50"></textarea></td>
                </tr>
                <tr>
                    <td></td>
                    <td><br>
                        <select name="active">
                            <option value="1">فعال</option>
                            <option value="0">غیرفعال</option>
                        </select>
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td><br><button type="submit" class="btn btn-
success"> ثبت </button></td>
                </tr>
            </table>
        </form>
    </div>
</body>
```

توجه دارید که بخش action از فرم ورود اطلاعات در نهایت به متد store ارجاع را انجام می دهد. همچنین چون store با حالت post کار می کند، method هم در فرم بر این اساس تعریف می شود. در ضمن توجه داشته باشید نامهایی که در بخش name داخل input داده می شوند باید دقیق مانند فیلدهای درون دیتابیس باشند.



بعد از این در فایل **categories.blade.php** هم می توان لینک ایجاد فهرست جدید را ویرایش کرد:

```
<a href="{{route('create')}}" class="btn btn-primary">دسته بندی جدید</a>
```

اما با ورود اطلاعات و زدن کلید ثبت خطای ۴۱۹ را دریافت خواهید کرد. دلیل امر ایجاد یک نکته ی امنیتی خواهد بود که به صورت تمهیدی برای جلوگیری از نفوذ[41] در نظر گرفته شده است.
به طور خلاصه حفره ی امنیتی CSRF[42] زمانی رخ می دهد که محتوای فرم ارسالی توسط شما را بتوانند با محتوای جدیدی عوض کرده و دوباره ارسال کنند! این موضوع می تواند زمان ارسال یک شماره کارت بانکی و یاحتا اطلاعات مهم دیگر بسیار خطرساز باشد. لاراول در پشت پرده ی کار خود برای رفع این مشکل، کلیدی را در مبدأ تعریف کرده و در مقصد همان را چک می کند. اگر هر دو یکی باشند به این معناست که اطلاعات فرم دستکاری نشده اند و اجازه ی دریافت را می دهد.
اما شما کل این مسیر را تنها با به کار بردن یک دستور درون برنامه ی خود انجام می دهید!
وقتی فرم شما از نوع ارسال POST استفاده می کند، کافیست بعد از دستور فرم **@csrf** را قرار دهید:

```
<form action="{{ route('store') }}" method="POST">
    @csrf
```

بعد به متد store درون کنترل هم رفته و محتوای آن را با dd پُر کنید:

[41] Hack
[42] Cross-site request forgery

```php
public function store( Request $request ) {
    dd($request);
}
```

دستور dd مخفف Die & Domp در واقع برای رصد بخش های مختلف کد در زبان PHP کاربرد دارد. به این معنی که ضمن قطع اجرا و در واقع کشتن برنامه در بخشی که قرار گرفته می تواند محتوای متغیرهایی را به عنوان پارامتر قبول کرده و به نمایش بگذارد.

حالا یک بار فرم ورود اطلاعات خود را بازخوانی[۴۳] کرده و دوباره کلید ثبت را بزنید:

```
Illuminate\Http\Request {#43 ▼
  #json: null
  #convertedFiles: null
  #userResolver: Closure($guard = null) {#216 ▶}
  #routeResolver: Closure() {#225 ▶}
  +attributes: Symfony\Component\HttpFoundation\ParameterBag {#45 ▶}
  +request: Symfony\Component\HttpFoundation\ParameterBag {#44 ▼
    #parameters: array:4 [▼

      "_token" => "H1UE0172ViyKwclIZw7PfMfcNf4DNP8CakZm7NNX"
      "title" => "اخبار فنی"
      "description" => "یک متن نمونه"
      "active" => "1"
    ]
  }
  +query: Symfony\Component\HttpFoundation\InputBag {#51 ▶}
  +server: Symfony\Component\HttpFoundation\ServerBag {#47 ▶}
  +files: Symfony\Component\HttpFoundation\FileBag {#48 ▶}
  +cookies: Symfony\Component\HttpFoundation\InputBag {#46 ▶}
  +headers: Symfony\Component\HttpFoundation\HeaderBag {#49 ▶}
```

اگر بخش request را باز کنید و به parameters بروید، محتوای ارسالی خود را به متد store خواهید دید. علاوه بر سه فیلدی که شما ارسال کرده اید، یک _token هم خواهید دید که در واقع همان کلید مربوط به csrf است و عملیات جلوگیری از هک شدن را هدایت می کند. مقدار و محتوای این کلید در هر بار اجرای فرم به صورت تصادفی[۴۴] و متفاوت ساخته می شود و همراه فرم ما ارسال می گردد. مشابه همین کد که به صورت کوکی[۴۵] در کامپیوتر کاربر ذخیره شده در سِرور توسط لاراول و متد store چک می شود. اگر در میانه ی راه فرم دستخوش تغییرات شود و به جای آن اطلاعات دیگری ارسال گردد، این کد در سِرور تأیید نشده و اجازه ی ذخیره صادر نمی گردد. این موضوع بسیار جدی است مخصوصاً زمانی که یک رُبات[۴۶] نرم افزاری بخواهد به صورت رگباری این فرم را پر کرده، ارسال کند و باعث اختلال در بانک اطلاعاتی ما شود.

به فایل **.env** که پیش از این در ابتدای کار برای تنظیمات اتصال به دیتابیس به آن رجوع کرده بودیم بروید و پارامتر زیر را در آن بیابید:

[43] Refresh
[44] Random
[45] Cookie
[46] Robot

SESSION_LIFETIME=120

این عدد به ثانیه مشخص می کند که زمان ارسال فرم چقدر باشد. همانطور که می بینید به طور پیشفرض دو دقیقه است و بعد از این زمان کلیدی که تعریف شده منقضی[47] می شود. یعنی بعد از این زمان هم اجازه ی ذخیره ی فرم داده نمی شود و باید صفحه ی فرم رفرش شود. این عدد را می توانید به دلخواه خود بیشتر کنید.

اما حالا برای اتمام فرآیند ذخیره به سراغ کنترلر و متد store می رویم:

```
public function store( Request $request ) {
   //dd($request);
   $category = new Category ([
    'title'=>$request->get('title'),
    'description'=>$request->get('description'),
    'active'=>$request->get('active'),
   ]);
   $category->save();
}
```

و همان طور که می بینید از همان پارامترهای request که از سمت فرم آمده بودند و با dd توانسته بودیم آنها را ببینیم، در اینجا استفاده کرده ایم و متغیر آرایه ای $category را با مدل Category که این مقادیر جدید را گرفته ساخته ایم. در نهایت کافیست که متد save را برای این مدل جدید صدا بزنیم تا با مقادیر جدید عمل ذخیره در بانک اطلاعات را انجام دهد.

اما باز هم برای ذخیره خطا خواهیم گرفت:

Illuminate\Database\Eloquent\MassAssignmentException
Add [title] to fillable property to allow mass assignment on [App\Category].

این موضوع به دومین مورد امنیتی بر می گردد. برای این منظور به شرح چند دستور مربوط به مدل می پردازیم. فایل مدل Category.php را در نظر بگیرید. این فایل با نام مدل Category که در داخل آن تعریف شده است، به صورت پیشفرض به جدولی در دیتابیس متصل خواهد شد که نامش categories باشد. اما در بخش آشنایی با مدل دیدیم که اگر نخواهیم اینطور شود و نام جدول ما فرق داشته باشد، مدل خود را به صورت زیر با دستور table تغییر می دهیم:

```
class Category extends Model
{
   protected $table = 'dastebandiha';
}
```

[47] Expire

که نام جدول مورد استفاده برای این مدل را از categories که به صورت پیشفرض در نظر گرفته می شود به dastebandiha تغییر می دهد. این موضوع در بخش ساخت مدل قبلاً به طور مفصل توضیح داده شده است.

اما خطایی که در آخرین مرحله ی ذخیره ی فرم، دریافت کرده اید هم یک نوع ویژگی[۴۸] مانند موارد ذکر شده است. یعنی شما با دستور زیر فیلدهای مورد نظر و مجاز خود را برای ذخیره باید تعیین کنید:

```
class Category extends Model
{
   protected $fillable = ['title','description','active'];
}
```

و به این ترتیب شما تعیین می کنید که چه فیلدهایی به طور دقیق مورد ذخیره قرار گیرند تا هیچ هکری نتواند به فیلدهای دیگر دسترسی داشته باشد یا از فرم شما برای دستکاری فیلدهای دیگر استفاده کند.

این بار دیگر با زدن کلید ثبت در فرم، یک صفحه ی سفید دریافت می کنید. اگر به phpMyAdmin و جدول categories هم مراجعه کنید متوجه خواهید شد که اطلاعات جدید ذخیره شده اند. برای دریافت پیام مناسب در صفحه، پس از ذخیره نیازمند مطالعه ی بخشهای بعدی این کتاب هستید.

اعتبارسنجی

اگر به اسناد لاراول در سایت و بخش Validation مراجعه کنید، به مثالهای بسیار جالب و متعددی برخورد خواهید کرد. بخش Available validation rules را که بسیار مفصل است مد نظر داشته باشید. اعتبارسنجی هایی که امکان ورود صحیح ایمیل، یا ورود فقط حروف یا ورود فقط عدد و حتا نمونه های ترکیبی را در آنجا بیابید.

نمایش مستقیم خطا

در مثال قبل درون متد store می توانیم بخش زیر را قبل از باقی دستورات اضافه کنیم:

```
$validateData = $request->validate([
    'title'=>'required|unique:categories|max:15',
    'description'=>'required',
]);
```

[48] Property

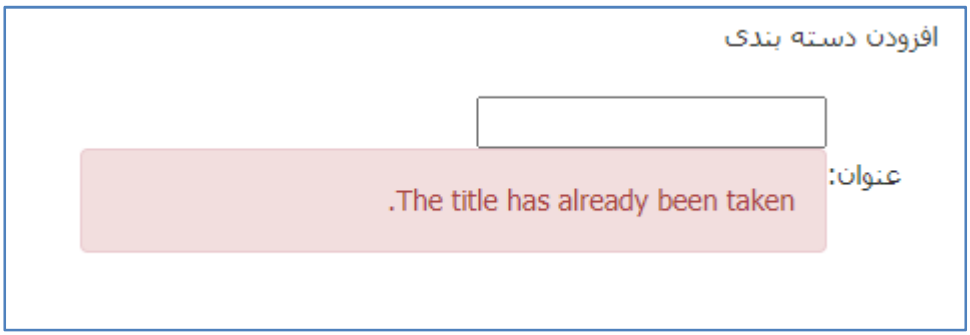
همانطور که می بینید سمت چپ نام فیلد و سمت راست انواع ولیدیشن نوشته شده اند که با علامت | از هم جدا شده اند. required یعنی این فیلد نمی تواند خالی ارسال شود. unique یعنی مقدار این فیلد نباید در دیتابیس تکراری بشود. max هم حداکثر طول کاراکترهای آن را تعیین می کند.

با همین کدهای جدید، دوباره برنامه را اجرا و فرم ورود اطلاعات را بیاورید. اگر در اینجا مقدار جدیدی وارد کنید، باز صفحه ی سفید را خواهید دید و با مراجعه به دیتابیس متوجه خواهید شد که مقادیر وارد شده، در بانک ذخیره شده اند. اما کافیست هنگام پر کردن فرم، مثلاً فیلد title یا عنوان را تکراری بنویسید. در این صورت با زدن دکمه ی ثبت، دیگر از صفحه ی سفید خبری نخواهد بود و باز به فرم ورود می گردید. همینطور اگر دیتابیس را چک کنید، متوجه خواهید شد که مقادیر تکراری درج نشده اند و این به دلیل اضافه کردن بخش اعتبارسنجی به برنامه بوده است.

حالا وقت آن رسیده که برای کاربر بتوانیم پیام مناسب را هم بعد از اعتبارسنجی فرم صادر کنیم. به سراغ فایل ویوی فرم یعنی **create.blade.php** می رویم:

```
<td align="left">عنوان:</td>
<td><br>
    <input type="text" name="title">
    @error('title')
     <div class="alert alert-danger">{{$message}}</div>
    @enderror
</td>
```

همانطور که روشن است در حال مدیریت خطا[۴۹] هستیم. دستور شروع خطا با @error و پایان آن با @enderror مشخص شده است. در پرانتز مقابل این دستور، خطایی را که برای فیلد title رخ داده را مد نظر داشته ایم. درون این دستور اما پیامی را نمایش می دهیم که در جریان خطای رخ داده به صورت پیشفرض درون متغیر $message ذخیره خواهد شد. با اجرای مجدد فرم ورود اطلاعات و دادن مقادیر تکراری در بخش عنوان، پیام خطا این بار به کاربر نشان داده خواهد شد:



[49] Error handeling

حالا اگر به جای مقدار تکراری، فیلد خالی ارسال کنید، با توجه به ولیدیشن required که برای عنوان به کار برده اید این بار خطای زیر به نمایش در می آید:

The title field is required

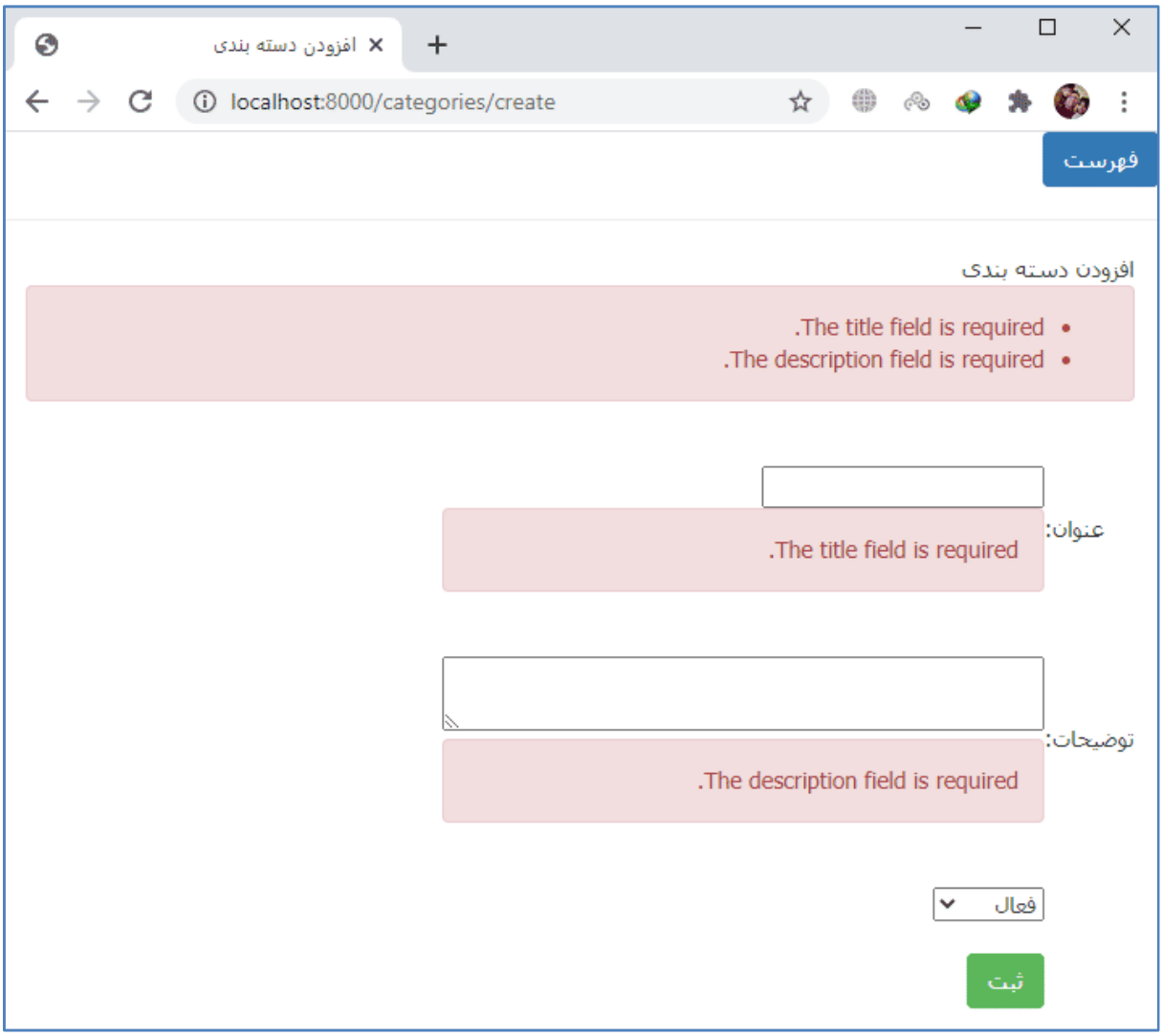
که یعنی نمی توانید مقدار خالی را به دیتابیس ارسال کنید. برای بخش description هم همین مدیریت خطا را می توانید انجام دهید:

```
<td align="left">توضیحات:</td>
<td><br>
    <textarea name="description" cols="50"></textarea>
    @error('description')
    <div class="alert alert-danger">{{$message}}</div>
    @enderror
</td>
```

همینطور می توانیم کاری کنیم که پیامهای خطا یک جا در بالای صفحه به نمایش در آیند. در این صورت باید در بالای صفحه و قبل از فرم پیام های مناسبی را برای ورود صحیح اطلاعات در فرم به نمایش بگذاریم. پس:

```
    <div class="container">
        {{ $pageTitle }}<br>
        @if ($errors->any())
            <div class="alert alert-danger">
                <ul>
                    @foreach ($errors->all() as $error)
                    <li>
                        {{$error}}
                    </li>
                    @endforeach
                </ul>
            </div>
        @endif
```

چون نمی دانیم چه تعداد خطا داریم از foreach استفاده کرده ایم. در شرط ابتدا چک می کنیم که خطایی رخ داده باشد. بعد از آن داخل شرط حلقه ای از تمام خطاها با متد ()all درست می کنیم و هر خطا را در متغیر $error جای داده و به صورت لیست نمایش می دهیم.
حالا اگر فرم را مثلاً خالی ارسال کنیم، متوجه می شویم که دو خطا در بالا قبل از فرم به نمایش در می آید و این علاوه بر پیامهایی است که کنار هر فیلد وجود دارد:



پس تا زمانی که خطایی وجود نداشته باشد، هیچ اخطاری به نمایش در نمی آید. با توجه به اینکه این کد در تمام فرم ها تکراری بوده و هر نوع خطایی را به نمایش می گذارد، می توان از تکرار آن جلوگیری کرد. برای این منظور در پوشه ی view پوشه ی جدیدی به نام layouts ایجاد کنید و داخل آن فایلی با نام **errors.blade.php** بسازید. همان بخش جدید کد را از فایل create.blade.php به کل برداشته و درون این فایل قرار دهید:

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

```
        </div>
    @endif
```

بعد از این به فایل create.blade.php برگشته و این فایل را به صورت زیر در همان بخش فراخوانی کنید:

```
<div class="container">
    {{ $pageTitle }}<br>
    @include('layouts.errors')
```

بعد از اجرا خواهید دید که نتیجه همان شکل قبل خواهد بود. در صفحات دیگر هم می توان با همین یک خط بخش نمایش خطاها را فعال کرد. توجه داشته باشید که در دستور include که به معنای شامل بودن است، مسیر پوشه و فایل با یک نقطه نمایش داده شده و از / خبری نیست. همینطور فقط نام ویو در آخر نوشته شده نه نام کامل فایل آن.

ترجمه ی خطاها

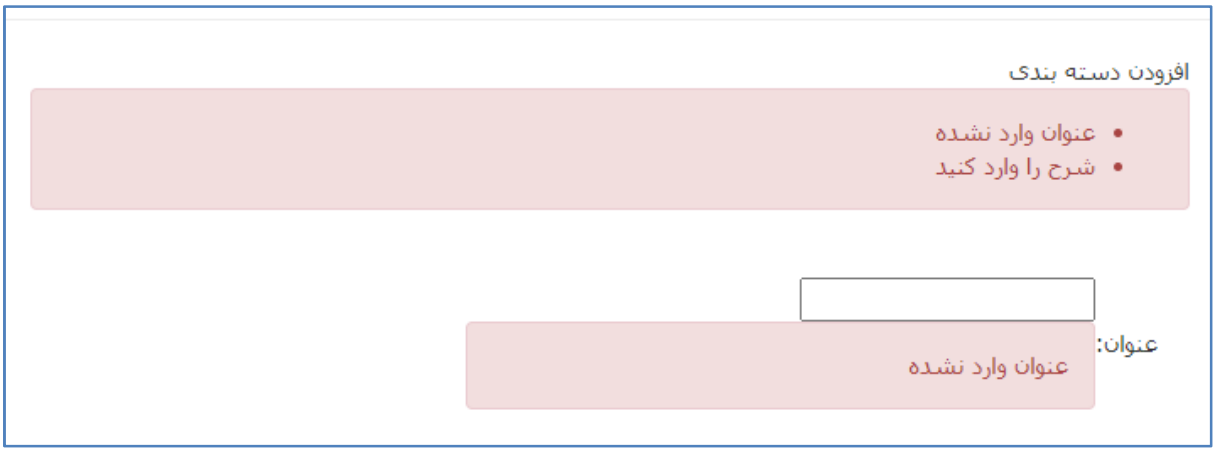
توجه کرده اید که این پیامها فارسی نیستند و برای نمایش در یک سامانه ی فارسی نمی توانند مناسب باشند. برای این منظور به بخشی باز می گردیم که در کنترلر، بخش ولیدیشن در آن فعال شده بود:

```
$validateData = $request->validate([
    'title'=>'required|unique:categories|max:15',
    'description'=>'required',
]);
```

در اینجا یک پارامتر دیگر را اضافه می کنیم تا به پیام پیشفرض که در $messages ذخیره می شود، محتوای جدیدی بدهیم. حالا تک تک پیامهایی را که برای هر ولیدیشن قرار است صادر شود را باید ترجمه کنیم. به این ترتیب که تمام ولیدیشن ها را در یک آرایه به نام $messages نوشته و مقابل آن معادل فارسی را قرار می دهیم:

```
public function store( Request $request ) {
    $messages = [
        'title.required'=> 'عنوان وارد نشده',
        'title.unique'=> 'عنوان تکراری است',
        'title.max'=> 'تعداد حروف مجاز ۱۵ کاراکتر',
        'description.required'=> 'شرح را وارد کنید',
    ];
    $validateData = $request->validate([
        'title'=>'required|unique:categories|max:15',
        'description'=>'required',
    ],$messages);
```

و پس از اجرا و ارسال مجدد فرم خالی، پیامهای خطا همه فارسی شده اند:



پیام ذخیره ی درست

نمایش پیام ذخیره ی اطلاعات در دیتابیس اما با فرض اینکه فعلاً همه چیز درست ذخیره شده است باید وجود داشته باشد تا به جای صفحه ی سفید، کاربر متوجه ی نتیجه بشود. روشن است که فرد باید به صفحه ی فهرست اصلی یعنی مسیر categories هم برگردد و موارد اضافه شده را در آنجا ببیند. پس یک انتقال نیز در این قسمت نیاز است. بنابراین در متد store پس از اجرای save انتقال را به مسیر مناسب انجام می دهیم:

```
        $category->save();
        return redirect(route('categories'));
    }
```

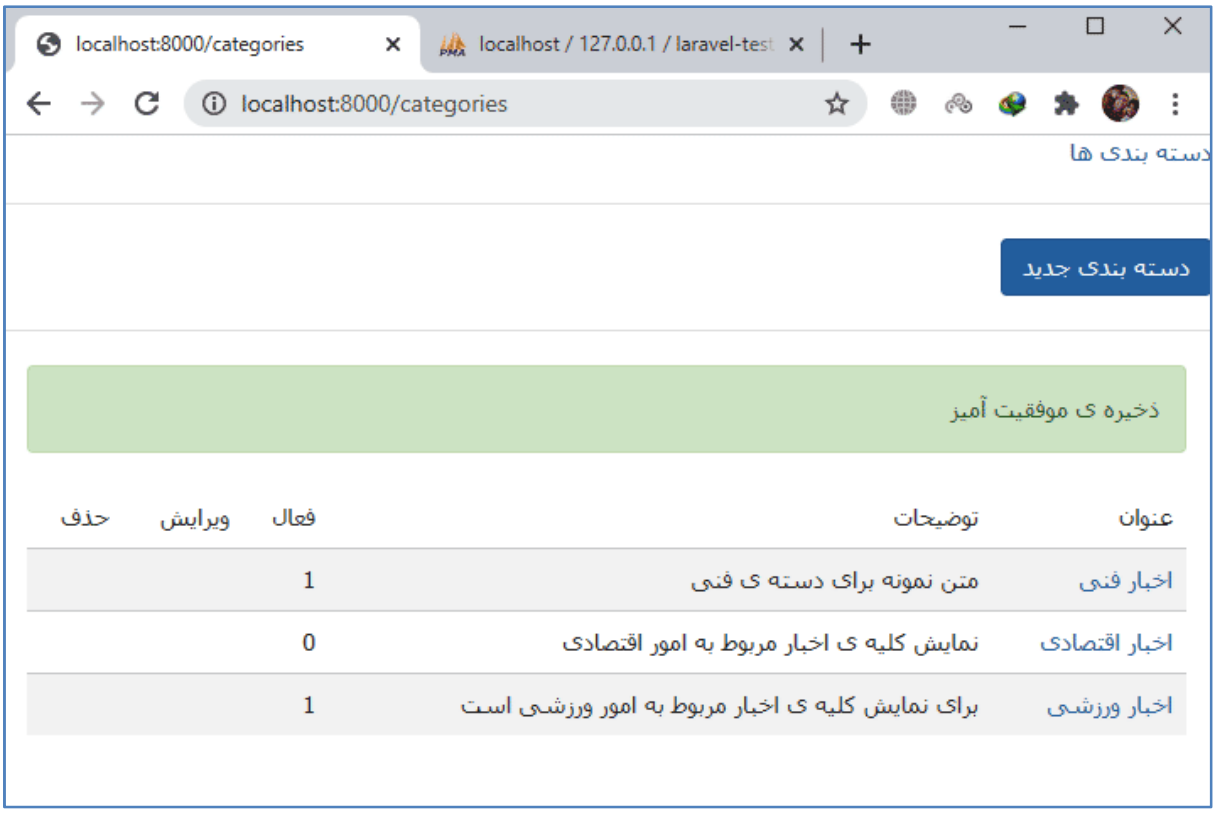
این بار بعد از ذخیره ی صحیح، دیگر به جای صفحه ی سفید به فهرست جدید دسته بندی ها باز می گردیم. اما هنوز پیامی به نمایش گذاشته نمی شود. بنابراین لازم است که قبل از انتقال به صفحه، پیامی را ساخته و در فرآیند انتقال به سمت صفحه ارسال کنیم:

```
        $category->save();
        $msg ="ذخیره ی موفقیت آمیز";
        return redirect(route('categories'))->with('success',$msg);
    }
```

حالا برای نمایش به سراغ ویوی نمایش فهرست یا categories.blade.php می رویم. در بالای جدول نمایش دسته بندی ها باید متغیر success را که به صورت یک سشن[50] یعنی متغیری که از صفحه ی دیگر آمده چک کنیم و اگر این متغیر مقدار داشته باشد، آن را نمایش دهیم:

```
<a href="{{route('categories')}}">
  {{ $pageTitle }}<br></a>
  <hr>
  <a href="{{route('create')}}" class="btn btn-primary">دسته بندی جدید</a>
  <hr>
  <div class="container">
      @if (session('success'))
      <div class="alert alert-success">{{session('success')}}</div>
      @endif
```

یک بار فرم را به طور صحیح پُر کرده و ارسال کنید:



و همانطور که می بینید فرآیند ذخیره و نمایش پیام مناسب پس از آن به طور کامل به اجرا در می آید. برای حالتی که ذخیره ناموفق باشد در بخش دیگری به آن خواهیم پرداخت. توجه داشته باشید که در اینجا نیز در صورت مراجعه ی مستقیم به لیست دسته بندی ها، پیام مذکور نمایش داده نشده و فقط زمانی که ذخیره ی فرم انجام شود این پیام دیده خواهد شد.

[50] Session

اما اگر خطایی به صورت استثناء[51] ایجاد شود، مثل زمانی که ذخیره به صورت ناموفق انجام شده باشد، نیازمند مدیریت از مسیرهای دیگری است. برای این منظور ابتدا کلاس Exception را باید در ابتدای کنترلر خود فراخوانی کنیم:

```
namespace App\Http\Controllers;

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

حالا پایین تر یعنی در متد store فرآیند save به هر دلیلی که با خطا مواجه شود توسط دستورات try و catch کنترل می شود:

```
try {
      $category->save();
    } catch ( Exception $exception ) {
        return redirect( route( 'categories' ) )->
with( 'warning', $exception->getCode() );
    }

    $msg = 'ذخیره موفقیت آمیز';
    return redirect( route( 'categories' ) )->with( 'success', $msg );
 }
```

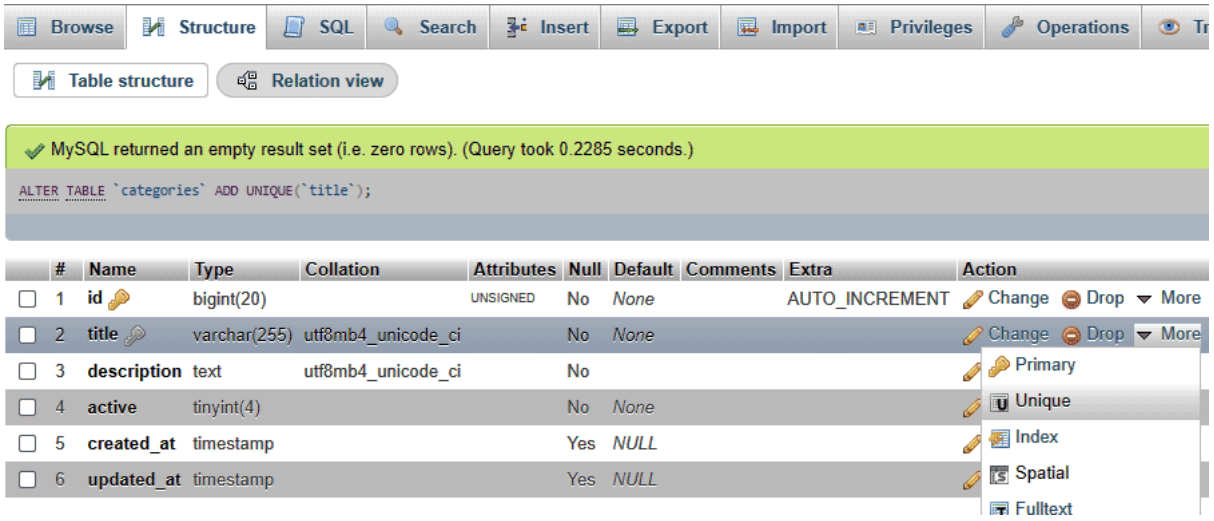
این بخش که در انتهای متد store قرا می گیرد ابتدا برای عملیات ذخیره سازی در بخش try تلاش لازم را می کند. اگر به هر دلیل موفقیت آمیز نباشد، به بخش catch رفته و کلاس خطای استثنایی یا Exception را فراخوانده و درون متغیر $exception می ریزد. کد خطای این مقدار همراه یک سشن به نام warning به سمت صفحه ی لیست دسته بندی های ارسال می گردد.

حالا می توانید به فایل خطای **errors.blade.php** مراجعه کرده و این خطا را نیز در آنجا کنترل می کنیم. یعنی خطوط زیر را در ادامه اضافه می کنیم:

```
@if (session('warning'))
<div class="alert alert-success">کد خطا: {{session('warning')}} </div>
@endif
```

و برای تست این بخش از طریق phpMyAdmin وارد ساختار جدول categories شده و فیلد title را با کلید More به حالت Unique می بریم:

[51] Exception



بعد از این کنترل خطای Unique را هم در بخش ولیدیشن کنترلر به صورت کامنت کرده و فقط حالت required را برای آن می گذاریم:

```
        $validateData = $request->validate( [
            'title'=>'required',
            // 'title'=>'required|unique:categories|max:15',
            'description'=>'required',
        ], $messages );
```

در فایل **categories.blade.php** هم فایل نمایش خطاها را فراخوانی می کنیم:

```
    <div class="container">
        @include('layouts.errors')
        @if (session('success'))
        <div class="alert alert-success">{{session('success')}}</div>
        @endif
```

در اینجا حالا که کنترل عنوان تکراری از داخل برنامه صورت نگرفته و از دیتابیس بررسی خواهد شد، در صورت ارسال عنوان تکراری پیام خطایی با مضمون زیر دریافت خواهید کرد:

کد خطا: ۲۳۰۰۰

که کد خطای عنوان تکراری در دیتابیس خواهد بود. برای این بخش هم میتوان ترجمه ای از پیامها را داشت:

```
        try {
            $category->save();
        } catch ( Exception $exception ) {
            switch ($exception->getCode()){
```

```
                case 23000:
                    $msg="عنوان تکراری";
                break;
            }
            return redirect( route( 'categories' ) )->with( 'warning', $msg );
        }
```

که به این ترتیب هر کد خطایی را می توان به عنوان یک کیس ترجمه و نمایش داد.

ویرایش اطلاعات

متد edit در کنترلر بسیار شبیه متد create خواهد بود با این تفاوت که فرم ارسالی خالی نیست و محتویات یک رکورد که از قبل موجود بوده را در خود داشته و با ذخیره محتویات آن را به روزرسانی می کند نه آنکه یک رکورد جدید ایجاد کند. همچنین مسیر بعد از create مسیری به نام store است و مسیر بعد از edit اما مسیر update خواهد بود که باید برنامه ریزی لازم در آن صورت گیرد. همانطور که در فایل web.php می توانید ببینید، فرق سوم edit با create در ساختار آدرسهای آنهاست که یک علاوه بر آدرس همنام خود، در حالت ویرایش یک id مربوط به رکورد مورد ویرایش هم در مسیر وجود دارد که به صورت متغیر {category} از مدلی به همین نام می آید.
و اما ساخت لینک در **categories.blade.php** هم برای ویرایش روش خاص خودش را دارد. چرا که باید کد رکورد مورد ویرایش، همراه لینک به متد edit ارسال گردد:

```
<body dir="rtl">
    @foreach($categories as $category)
        <tr>
            <td>
              <a href="{{route('show',$category->id)}}">
              {{ $category->title }}
              </a>
            </td>
            <td>{{ $category->description }}</td>
            <td>{{ $category->active }}</td>
            <td>
            <a href="{{route('edit',$category->id)}}"> ویرایش </a>
            </td>
            <td></td>
        </tr>
    @endforeach
```

که به این ترتیب لینک های مربوط به ویرایش هر سطر فعال می شوند. به سراغ کنترلر و متد edit رفته و به سادگی خطوط فراخوانی ویوی ویرایش را می نویسیم:

```
public function edit( Category $category ) {
```

```
        $pageTitle = 'ویرایش';
        return view( 'edit', compact( 'pageTitle', 'category' ) );
    }
```

حالا دیگر شروع به ساختن فرم ویرایش می کنیم. فایل **edit.blade.php** را در مسیر views بسازید:

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="../../bootstrap.min.css">
    <title>{{ $pageTitle }}</title>
</head>

<body dir="rtl">
    <a class="btn btn-primary" href="{{ route('categories') }}">
        فهرست<br>
    </a>
    <hr>
    <div class="container">
        {{ $pageTitle }}<br>
        @include('layouts.errors')
<form action="{{ route('update',$category->id) }}" method="POST">

            @method('put')
            @csrf
            <table>
                <tr>
                    <td align="left"> عنوان: </td>
                    <td><br>
<input type="text" name="title" value="{{ $category->title }}">
                        @error('title')
                        <div class="alert alert-danger">{{ $message }}</div>
                        @enderror
                    </td>
                </tr>
                <tr>
                    <td align="left"> توضیحات: </td>
                    <td><br>
<textarea name="description" cols="50">{{ $category->description }}</textarea>
                        @error('description')
                        <div class="alert alert-danger">{{ $message }}</div>
                        @enderror
                    </td>
                </tr>
                <tr>
                    <td></td>
```

```
                    <td><br>
                        <select name="active">
<option value="1" <?php if ($category->active == 1) {echo 'selected';} ?>>
 فعال </option>
<option value="0" <?php if ($category->active == 0) {echo 'selected';} ?>>
 غیرفعال </option>
                        </select>
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td><br>
<button type="submit" class="btn btn-success"> ویرایش </button>
                    </td>
                </tr>
            </table>
        </form>
    </div>
</body>
```

در این بخش به قسمت های متفاوت با فرم create و مخصوصاً روش ساختن کومبوباکس انتخاب برای حالت های فعال و غیرفعال توجه کنید. اینکه چطور آخرین وضعیت فیلد active را می توان خواند و با یک شرط به option منتقل کرد.

همینطور توجه کنید به دستور @method که در ابتدای فرم تعریف شده است. با توجه به اینکه در بین پروتکل های موجود برای ارسال فرم، چیزی به نام put در اصل وجود خارجی نداشته و جزو اختراعات لاراول است، امکان تعریف آن در form هم نیست. بنابراین در فرم همان متد post قرار داده می شود. اما این put همان متدی است که در بخش مسیردهی به مسیر بعد از فرم ویرایش یعنی متد update اختصاص داده شده است.

به سراغ متد update در کنترلر رفته و همان مقادیر درون store را با اندکی تغییرات در آن وارد می کنیم:

```
    public function update( Request $request, Category $category ) {
        $messages = [
            'title.required'=> 'عنوان وارد نشده',
            'title.unique'=> 'عنوان تکراری',
            'title.max'=> 'تعداد حروف مجاز ۱۵ کاراکتر',
            'description.required'=> 'شرح وارد نشده',
        ];
        $validateData = $request->validate( [
            'title'=>'required',
            'description'=>'required',
        ], $messages );

        $category->title = $request->title;
        $category->description = $request->description;
```

```
        $category->active = $request->active;

        try {
            $category->save();
        } catch ( Exception $exception ) {
            switch ( $exception->getCode() ) {
                case 23000:
                $msg = 'عنوان تکراری';
                break;
            }
            return redirect( route( 'categories' ) )->with( 'warning', $msg );
        }

        $msg = 'ویرایش موفقیت آمیز';
        return redirect( route( 'categories' ) )->with( 'success', $msg );
    }
```

به قسمت های ولیدیشن دست نخورده است. تنها مقادیر آبجکت درون $category باید با مقادیر ارسال شده از طرف فرم و در $request جابجا شوند. بعد از آن دیگر مراحل ذخیره مانند متد store انجام خواهد شد.

حذف اطلاعات

همانطور که از قبل دیده اید، این آخرین متد در کنترلر است که در بخش destroy نوشته می شود. به فایل web.php که مسیرها درون آن تعریف شده اند باز توجه کنید. مسیر categories/destroy هم یک id مربوط به رکورد انتخابی را مانند بخش ویرایش گرفته و بعد اقدام به حذف رکورد مربوطه خواهد کرد. اول از همه لینک مربوط به حذف را با توجه به مسیر مرتبط، کنار لینک ویرایش در صفحه ی **categories.blade.php** ایجاد می کنیم:

```
<a href="{{ route('edit', $category->id) }}"> ویرایش </a>
</td>
<td>
    <a href="{{ route('destroy', $category->id) }}"> حذف </a>
</td>
```

و حالا وقت آن است که به سراغ کنترلر برویم:

```
  public function destroy( Category $category ) {
        $category->delete();
    }
```

بله همین یک خط! یا بهتر است بگوییم یک دستور، خودش همه کار می کند. با اجرای برنامه و فشردن لینک **حذف** در مقابل رکورد دلخواه، باز صفحه ی سفید می آید و اگر به ویوی categories هم برگردیم متوجه ی حذف آن رکورد خواهیم شد. بله این ساده ترین بخش نسبت به بخش های دیگر بود. بهتر است فرآیند کار خود را تکمیل کنیم. یعنی ریدایرکت یا انتقال به لیست از کنترلر:

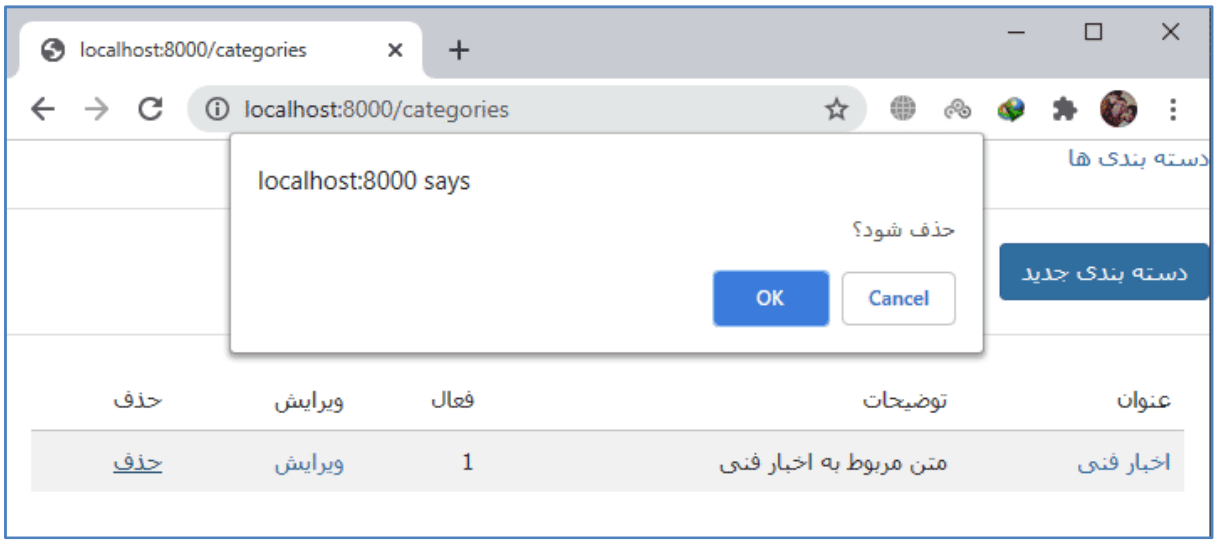
```
public function destroy( Category $category ) {
  $category->delete();
  $msg = 'حذف موفقیت آمیز';
return redirect( route( 'categories' ) )->with( 'success', $msg );
}
```

و حالا همه چیز آنطور که باید کار می کند. به مجموعه ی چهار عمل اصلی کار با اطلاعات که در آن ایجاد، خواندن، ویرایش و حذف اتفاق می افتد، CRUD[52] گفته می شود.
اگر دقت کرده باشید باز روال حذف و فرآیندی که برای نابود کردن دائمی یک رکورد اطلاعاتی طی کرده ایم، به طور معمول منطقی نیست. یعنی همیشه قبل از حذف باید یک پرسش برای تأیید[53] عمل حذف وجود داشته باشد. تنها با یک خط کد جاوااسکریپت[54] ساده این عمل را انجام می دهیم. زبانی که قبل از ارسال صفحات کنترل هر رویداد[55] را بر عهده می گیرد:

```
<a href="{{ route('destroy', $category->id) }}"
    onclick="return confirm('حذف شود؟');">
        حذف
</a>
```

و قبل از حذف سوال می شود که با زدن کلید Ok حذف انجام می شود:

[52] Create, Read, Update, Delete
[53] Confirm
[54] Javascript
[55] Event



با توجه به اینکه دستورات جاوااسکریپت سیستمی هستند، کلیدهای جعبه ی گفتگو[56] قابل ترجمه نیستند و می توانید از مرورگر نسخه ی فارسی برای نمایش فارسی این کلیدها استفاده کنید. تنها مطلب باقیمانده مدیریت خطای حذف در کنترلر است:

```
public function destroy( Category $category ) {
        try {
            $category->delete();
        } catch ( Exception $exception ) {
return redirect( route( 'categories' ) )->
with( 'warning', $exception->getCode() );
        }
        $msg = 'حذف موفقیت آمیز';
return redirect( route( 'categories' ) )->with( 'success', $msg );
    }
```

به این ترتیب در صورت بروز خطا هنگام پاک کردن از دیتابیس، این خطا به لیست ارسال می گردد.

احراز هویت

تا لحظه ی نگارش این متن یعنی تابستان سال ۲۰۲۰ میلادی که نسخه ی ۷ لاراول منتشر شده، این فریمورک نسخه های متعدد با قابلیت های گوناگون داشته است. به عنوان مثال در نسخه ی ۵ بخش احراز هویت[57] برای کاربرانوجود داشته است. ولی در نسخه ی ۷ به دلیل تعدد امکانات و ابزارها، انتخاب نصب آن بر عهده ی برنامه نویس گذاشته شده و هسته ی اصلی لاراول را این

[56] Dialogue box
[57] Authentication

بخش همراهی نمی کند. این حالت به فریمورک ها و کتابخانه های کد کمک می کند که به صورت پیشروَنده[58] عمل کنند. یعنی بدون اینکه بی جهت حجم هسته ی اصلی کتابخانه زیاد باشد، فقط زمانی که یک ماژول[59] لازم باشد، آن را بارگیری[60] و استفاده می کنند. به این ترتیب می شود انواع مختلفی را حتا برای یک نوع کار و ماژول طراحی کرد و حق انتخاب بیشتری داشت.

نصب ماژول احراز هویت

یک پروژه ی خام لاراول را آنطور که ابتدای این مجموعه فرا گرفتیم، نصب و آماده ی کار کنید. چهره ی صفحه ی اول پروژه ی خام را به خاطر بیاورید. هیچ جایی برای ثبت نام یا ورود کاربران در آن وجود نداشت. البته فایلی به نام User.php در مسیر app وجود دارد که از سه بخش معماری MVC در واقع این بخش مدل برای ارتباط با ویو و کنترلرهای کار با کاربران است. بعد از اینکه مطمئن شدید تنظیمات دیتابیس در فایل env. درست انجام شده و دیتابیس پروژه ساخته شده است، دستور زیر را اجرا کنید:

php artisan migrate

جدول users و فیلدهای خاص آن همراه سه جدول دیگر ساخته خواهد شد. برای کار با بخش احراز هویت کاربران ابتدا دستور زیر را اجرا کنید، تا فایل های لازم دانلود شوند:

composer require laravel/ui

بعد از اتمام کار این دستور، دستور زیر را برای نصب این بخش در مسیر پروژه اجرا کنید:

php artisan ui vue --auth

به این ترتیب تغییرات واضحی در پروژه رخ خواهد داد. مهمترین چیز اضافه شدن دو پوشه ی Auth یکی در مسیر views و دیگری در مسیر Controllers است. همچنین مسیری در فایل web.php به صورت زیر تعریف شده است:

```
Auth::routes();
```

که در واقع مسیرهای مربوط به احراز هویت را فراخوانی می کند. حالا با اجرای پروژه تصویر زیر را خواهیم داشت:

[58] Progressive
[59] Module
[60] Download

گوشه ی بالا سمت راست را نگاه کنید. بخش های LOGIN و REGISTER هم اضافه شده اند. به سراغ فایل app.blade.php بروید. بیشتر تنظیمات ظاهری صفحات بخش احراز هویت در این بخش صورت می گیرد. به عنوان مثال می توانید با افزودن لینک های مناسب ظاهر صفحات را با فریمورک Bootstrap اصلاح کرده یا صفحات را راست چین کنید. همچنین از پوشه ی views این صفحات را پیدا کرده و به فارسی ترجمه کنید.

در پوشه ی Http بعد از Controllers پوشه ای به نام Middleware یا میان افزار ساخته شده است. کاربرد میان افزارها در احراز هویت بسیار اساسی است. به این معنی که می توان به کمک آنها دسترسی کاربر را به بخش های مختلف برنامه تعریف کرد.

برای امتحان، یک مسیر خیلی سریع به صورت زیر تعریف کنید:

```
Route::get('/show', function () {
    return view('show');
})->middleware('auth');
```

بخش آخر مسیر به طور دقیق تعیین می کند که فقط کاربرانی قادر به مشاهده ی این ویو هستند که لاگین انجام داده باشند. اگر فایل ویوی این مسیر یعنی **show.blade.php** را هم بسازید خواهید دید که تا زمانی که به عنوان یک کاربر شناخته شده، رجیستر نکرده و به سیستم وارد نشده باشید، قادر به مشاهده ی این ویو نیستید. به همین سادگی برای تمام صفحات خود می توانید ابتدایی ترین دسترسی یعنی نمایش بعد از ورود را ایجاد کنید. فقط برای رجیستر سریع و ساختن یک کاربر آزمایشی توجه داشته باشید که بعد از ورود ایمیل و نام کاربر، رمز عبور باید حداقل ۸ کاراکتر داشته باشد.

اگر به هر دلیلی ورود انجام نشده باشد ولی یک مسیر را در نوار آدرس مرورگر صدا بزنیم یا لینکی را کلیک کنیم که اجازه ی نمایش آن قبل از ورود داده نشده است، به طور خودکار کاربر به صفحه ی لاگین منتقل خواهد شد. حالا فایل صفحه ی اصلی یا **welcome.blade.php**را نیز نگاهی بیاندازید:

```
<body>
    <div class="flex-center position-ref full-height">
        @if (Route::has('login'))
            <div class="top-right links">
                @auth
                    <a href="{{ url('/home') }}">Home</a>
                @else
                    <a href="{{ route('login') }}">Login</a>

                    @if (Route::has('register'))
                        <a href="{{ route('register') }}">Register</a>
                    @endif
                @endauth
            </div>
        @endif
```

همانطور که مشاهده می کنید به کمک یک شرط ابتدا مسیر لاگین آزمایش می شود. اگر شرایط در حالت ورود موفق باشد محتوای بعد از @auth یعنی لینک صفحه ی خانه به نمایش در می آید. به این ترتیب لینک های LOGIN و REGISTER دیگر نمایش داده نمی شوند. اما اگر ورود ناموفق بوده یا انجام نپذیرد، لینک های مذکور که در بخش بعد از @else تعریف شده اند به نمایش در می آیند.

نقش های کاربری

تا این لحظه فقط توانستیم نقش دسترسی کاربر وارد شده یا وارد نشده را ایجاد کنیم. در واقع فقط یک نقش برای دسترسی داریم که می تواند مدیر سیستم باشد و برای نقش های دیگری که به برخی بخش و اطلاعات دسترسی نداشته باشد کاری نکرده ایم.

برای این منظور قبل از هر چیز یک فیلد نقش یا role باید به جدول کاربران خود اضافه کنید. راحت ترین و مطمئن ترین روش، ویرایش یکی از مایگریشن های پروژه است که با نام _create_users_table در انتهای نام فایلش ساخته شده است. این فایل را باز کرده و فیلد role را به آن اضافه می کنیم:

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

البته در اینجا دستور اجرای مایگریشن در صورتی کار خواهد کرد که قبلاً اجرا نشده باشد. اگر اینطور نبود با توجه به مطالبی که در بخش آموزش ساخت مایگریشن توضیح داده ایم، اقدام به ساخت یک مایگریشن جدید برای این فیلد نمایید. تنها نکته ی مهم در این بخش آن است که فیلد role با مشخصات بالا در جدول users موجود باشد.

روشن است که ما تنها یک نوع middleware برای ورود کاربران داشتیم. پس باید نمونه ی اختصاصی خود را بسازیم. پس در خط فرمان ترمینال خواهیم داشت:

php artisan make:middleware CheckRole

بعد از ساخت این فایل آن را که در پوشه ی Middleware با نام **CheckRole.php** قرار دارد به صورت زیر ویرایش می کنیم:

```
    public function handle($request, Closure $next)
    {
        if(auth()->check() && auth()->user()->role ==1 ){
            return $next($request);
        }
        return redirect(route('login'));
  }
```

پس از این ما چک کرده ایم که اگر مقدار فیلد role برابر عدد یک باشد، یعنی کاربری که وارد شده است کاربر خاص یا ادمین است. به این ترتیب دسترسی های ویژه ای را به او خواهیم داد که به دیگر کاربران وارد شده نخواهیم داد. همچنین اگر با توجه به این دسترسی اگر کاربر مورد نظر شرط لازم را برای دسترسی نداشت به صفحه ی login فرستاده شود.

این میدل ویر جدید که نوشته ایم باید به شکلی رجیستر یا ثبت در پروژه باشد. برای این منظور فایل **Kernel.php** را از پوشه ی app باز کرده و در انتهای آن به همراه میدل ویرهای دیگر نام

این نمونه ی جدید خود را اضافه کنید. در واقع ما با بخش routeMiddleware که مربوط به انواع قابل نسبت دادن به مسیرهاست کار داریم:

```
protected $routeMiddleware = [
'auth' => \App\Http\Middleware\Authenticate::class,
'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
'can' => \Illuminate\Auth\Middleware\Authorize::class,
'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
'checkrole' => \app\Http\Middleware\CheckRole::class,
```

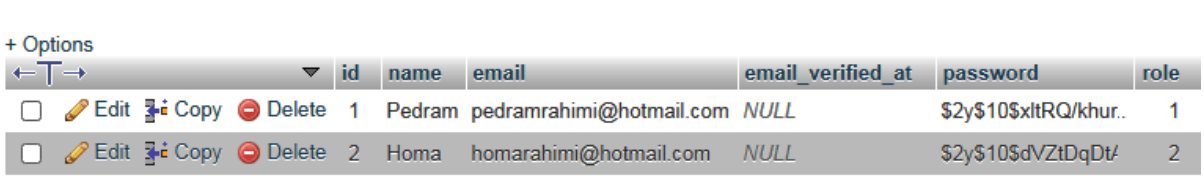
در واقع از مسیر یاد شده کلاسی با همین نام فراخوانی خواهد شد. حالا توجه داشته باشید بر اساس چیزی که در بخش ساخت فرم ورود اطلاعات فرا گرفته اید، مدل **User.php** و قسمت های fillable را هم اصلاح کنید تا بخش role قابل ثبت باشد:

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

    protected $attributes = [
        'role' => 2,
    ];
```

بخش attributes را برای ذخیره ی مقدار پیشفرض در نقش کاربر به صورت ۲ نوشته ایم تا هیچ کاربری در حالت عادی کاربر ۱ که قرار است مدیر سیستم باشد ساخته نشود.

حالا بعد از ذخیره ی تغییرات به سراغ فرم رجیستر یا ثبت نام کاربران بروید. دو کاربر مختلف بسازید. و در دیتابیس چک کنید که مقدار role برای هریک ۲ ذخیره شده است.

به صورت دستی role را برای کاربر اول در دیتابیس تغییر داد و مقدار یک بدهید.



و آخرین کار این است که مسیر نمونه ای را که برای نمایش کاربران وارد شده به نام show در مثال های قبلی ساخته بودیم این بار با میدل ویر جدید خود در فایل **web.php** بسنجیم:

```
Route::get('/show', function () {
    return view('show');
})->middleware('checkrole');
```

حالا فقط زمانی که با کاربر نقش یک بخواهید این صفحه را ببینید به شما اجازه داده خواهد شد.همچنین می توانید فرم های ویرایش جدول کاربران را با دسترسی نقش یک یا مدیر سیستم بسازید که در صورت نیاز به کاربران دیگر بتوانید از داخل برنامه دسترسی مدیر داده و نقش آنها را ویرایش کنید.

در نظر داشته باشید که با مطالعه ی پکیج های ساخته شده برای لاراول در اینترنت می توانید به موارد بسیاری برخورد کنید که بخش دسترسی ها را نیز به طور خودکار پیاده سازی می کنند و شما فقط باید مانند خود بخش احراز هویت، اقدام به نصب این ماژول ها بر روی پروژه ی خود کنید.

آپلود تصاویر

ابتدا مسیرهای زیر را بسازید:

```
Route::get('image-upload', 'ImageUploadController@imageUpload')-
>name('image.upload');
Route::post('image-
upload', 'ImageUploadController@imageUploadPost')-
>name('image.upload.post');
```

یک کنترلر با نام ImageUploadController ساخته و محتوای زیر را به آن بدهید:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ImageUploadController extends Controller
{
    /**
```

```
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function imageUpload()
    {
        return view('imageUpload');
    }

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function imageUploadPost(Request $request)
    {
        $request->validate([
            'image' =>  'required|image|mimes:jpeg,png,jpg,gif,svg|max:2048',
        ]);
        $imageName   =   time().'.'.$request->image->extension();
        $request->image->move(public_path('images'), $imageName);

        return back()
            ->with('success','آپلود موفقیت آمیز بود')
            ->with('image',$imageName);
    }
}
```

در بخش mimes می توانید انواع فایل های قابل دریافت و بیشترین اندازه ی فایل را در پارامتر max به واحد بایت مشخص کنید.

فایل imageUpload.blade.php را هم در قسمت ویوها بسازید:

```
<!DOCTYPE html>
<html>
<head>
    <title> آپلود عکس در لاراول </title>
    <link rel="stylesheet" href="bootstrap.css">
    <style>
        body{
            direction: rtl;
            text-align: right;
        }
    </style>
</head>
```

```
<body>
<div class="container">

    <div class="panel panel-primary">
        <div class="panel-heading"><h2>آپلود عکس </h2></div>
        <div class="panel-body">

            @if ($message = Session::get('success'))
                <div class="alert alert-success alert-block">
                    <button type="button" class="close" data-
dismiss="alert">×</button>
                    <strong>{{ $message }}</strong>
                </div>
                <img src="images/{{ Session::get('image') }}">
            @endif

            @if (count($errors) > 0)
                <div class="alert alert-danger">
                    <strong>وای!</strong> مشکلی رخ داد
                    <ul>
                        @foreach ($errors->all() as $error)
                            <li>{{ $error }}</li>
                        @endforeach
                    </ul>
                </div>
            @endif

            <form action="{{ route('image.upload.post') }}" method="POST" enct
ype="multipart/form-data">
                @csrf
                <div class="row">

                    <div class="col-md-6">
                        <input type="file" name="image" class="form-control">
                    </div>

                    <div class="col-md-6">
                        <button type="submit" class="btn btn-
success">بارگزاری</button>
                    </div>

                </div>
            </form>

        </div>
    </div>
</div>
</body>
```

```
</html>
```

حالا پس از ذخیره ی این فایل ها در نوار آدرس مرورگر به نشانی image-upload رفته و نتیجه را چک کنید. پس از آپلود تصاویر، در پوشه ی public پوشه ای به نام images خواهید داشت که تصاویر در آنجا قرار می گیرند. توجه داشته باشید که این کد بسیار خلاصه بوده و یک مدیریت کامل فایل برای ذخیره ی نشانی تصویر در دیتابیس یا حذف فایل بعد از آپلود نیست. بنابراین توصیه می شود که به جای آن از نمونه کدهای آماده برای مدیریت فایلهای خود استفاده کرده و برای نوشتن آن دچار زحمت نشوید.

پایان

تنها مرجع این کتاب سایت اینترنتی Laravel.com است.