Yeni Başlayanlar için
LARAVEL

Umut Doğru

Önsöz

Değerli okurum,

Günümüz dünyası hızla değişiyor, bu gerçeğin farkındayız; ancak, hayatı bir tren yolculuğu gibi düşünürsek ve eğer bu hızlı treni kaçırırsak, kendimizi yerimizde sayarken bulabiliriz. İşte bu yüzden yapmamız gereken, o treni kaçırmadan atlayıp, değişim ve yeniliklerle birlikte ilerlemek.

Söz konusu olan tren, günümüz teknolojisinin getirdiği imkanlarla donatılmış olan web teknolojileridir. İşte bu kitap da, bu önemli trenin en değerli vagonlarından birine odaklanıyor: Laravel framework'u. Günümüzün birçok web geliştiricisinin tercih ettiği Laravel, esnek, modern ve güvenli programlama kapılarını sizlere aralıyor; ancak, belirtmeliyim ki bu trenin hareketine katılmadan önce belirli bir teknoloji bilgisine ulaşmış olmanız gerekiyor.

Kitabın içeriğinde direkt olarak belirtilmese de, Laravel bir framework olarak belirli bir temel bilgi düzeyini gerektiriyor; ancak, şunu unutmamanız önemlidir: Kitabı bitirdiğinizde, **"Artık her şeyi biliyorum"** demek yerine asıl hedefiniz kitap sonrası dönemdir. Sürekli kendinizi geliştirmek, bilginizi derinleştirmek gereklidir. Bu noktada, daha fazla makale okumak, uzman kişilerin videolarını izlemek ve en nihayetinde kendi projelerinizi geliştirmek büyük önem taşır. Bu adımları takip ederseniz, bir süre sonra bu alandaki gelişiminizi net bir şekilde gözlemleyeceksiniz.

Teknolojinin hızla evrildiği bu çağda her adımınızın sizleri geleceğe daha da yakınlaştırdığını unutmayın. Sözlerimi Gazi Mustafa Kemal Atatürk'ün şu özlü sözü ile noktalayarak tamamlamak istiyorum:

"Gençler, cesaretimizi takviye ve idame eden sizlersiniz. Siz, almakta olduğunuz terbiye ve irfan ile insanlık ve medeniyetin, vatan sevgisinin, fikir hürriyetinin en kıymetli timsali olacaksınız. Yükselen yeni nesil, istikbal sizsiniz. Cumhuriyeti biz kurduk, onu yükseltecek ve yaşatacak sizsiniz."

2023, Umut DOĞRU

İstanbul - Türkiye

Değerli okurlarım kitabımıza başlamadan önce bir ricam olacak.

Kitapta hata gördüğünüzde hemen kızmayın, sizlerin de desteğiyle hatalarımızı ve eksikliklerimizi tarafıma bildirmeniz durumunda bir sonraki baskıda düzeltilecektir. :)

İçindekiler

1

Giriş

Bu Bölümde

Bu bölümde, MVC yapısı hakkında bilgi sahibi olacak ve MVC mimarisi olan Laravel'in kurulumu için gerekli olan aşamaları işletim sistemleri için birlikte kurulumunu öğreneceğiz.

MVC NEDİR?

MVC (Model View Controller) için bir kısaltmadır. Tanımlamak gerekirse yazılım mühendisliğinde kullanılan bir **Mimari Desen** veya **Tasarım Deseni**'dir. Geliştiricilerin uygulama oluştururken benimsedikleri mimariyi temsil eder. MVC mimarisiyle veri akışının nasıl işlediğini kontrol edebiliyoruz.

MVC üç bileşenden oluşmaktadır bu işlemi 3 ayrı katman ile yapar.

Model | View | Controller

» **Model,** uygulamanın temel davranışlarını ve verilerini yönetir. Bilgi talepleri üzerine sana yanıt verebilir, akışın durumunu değiştirebilir hatta sizi bilgilendirebilir. Bu bir veritabanı, veri yapısı veya depolama sistemi olabilir. Kısacası uygulamanın veri yönetiminden sorumludur.

» **View,** uygulamanın kullanıcı arabirimi tarafını yönetir. Modelden gelen verileri uygun bir forma dönüştürür.

» **Controller,** kullanıcı verisini alır ve eylemleri doğru bir şekilde göndermek için nesneleri ve görünümü hazırlar.

Basit bir senaryoya bakalım:

Bir e-ticaret web sitesine giderseniz; gördüğünüz farklı sayfalar View katmanı tarafından sağlanır.

Daha fazlasını görüntülemek için belirli bir ürüne tıkladığınızda, Controller katmanı kullanıcının eylemini işler.

Bu, Model katmanını kullanarak bir veri kaynağından veri almayı içerebilir. Veriler daha sonra bir araya getirilir ve View katmanında düzenlenir ve kullanıcıya görüntülenir. Sonuç olarak üç bileşen MVC yapısını yönetmek için kolay bir yol sunar.

NEDEN MVC?

PHP uygulamaları oluştururken birden fazla dosya ile çalışmak sorun olmayabilir hatta kullanımı çok kolaydır. Çünkü kendisi bir mimari ile yönetiliyor ve bu da bir yapı ile devamlılık sağlıyor.

Mimarisi olmayan kod tabanları ile çalışmak zorunda kaldığınızda özellikle proje büyükse bütün kodları tek bir yerde yığın halinde bırakmak işi çok zorlaştırır. Kodunuz belki bir hafta belki bir ay çalışır. Eğer ki ekleme yapmak ya da bir şeyleri değiştirmek isterseniz bunu yapmadığınız taktirde güncelleme olduğunda projeniz patlar yani çalışmaz.

Daha teknik bir notta, MVC mimarisini kullanarak inşa ettiğinizde aşağıdaki stratejik avantajlara sahip olursunuz:

» Projede bulunan bileşenleri parçalamak çok daha kolaydır.

o MVC mimarisine sahip olduğunuzda projedeki bileşenleri bölmeniz çok kolay olacaktır. Bir Front-End geliştiricisi olup view'lar ile çalışabilirsiniz. Eğer ki Back-End tarafında çalışan farklı birisi var ise bu işin belli dosyalarını ona verip işi daha da geliştirebilirsiniz. Bu işlem işi çok kolay bir hale getirecektir.

» Sorumluluk İzolasyonu!

o Bunu şöyle anlatabilirim: Örneğin; Model ile View birbirine bağlı değildir. Bu yüzden Model'de yaptığınız bir değişiklik View katmanını etkilemeyecektir.

» Uygulama URL'nin tam kontrolü!

o MVC mimarisi ile proje içerisinde bulunan route ile tam kontrole sahip olursunuz. Bu SEO tarafında işinizi kolaylaştırır.

MVC sisteminin bu prensipleri çok basittir. Konuyu kavradığınızı düşünüyorum.

Ortam Kurulumları ve Composer

Windows için Kurulum

Editör Kurulumu

İlk olarak editörü kurmamız gerekiyor. Bu konuda Laravel yazdığımızı düşünürsek kullanabileceğiniz en iyi IDE **PhpStorm** olacaktır.

https://www.jetbrains.com/phpstorm/

İndirme linkine gittiğinizde 30 günlük bir deneme sürümü ile karşılaşacaksınız. Daha sonrasında sizden ücret talep edecek.

Eğer ki bu ücreti vermek istemiyorsanız alternatif olarak **Visual Studio Code** editörünü de kullanabilirsiniz.

https://code.visualstudio.com/ adresinden kurulumunu yapabilirsiniz.

Terminal Kurulumu

Laravel yazarken bazı paketleri kurmak veya araçlara erişmek için terminal kullanırız. Kendisi bizim için önemlidir. Windows için olabilecek en kapsamlı terminal **Tabby**'dir. Terminalin kendi web sitesinden kurulumu tamamlayabilirsiniz. ***https://tabby.sh***

Sitenin **Download** kısmına tıkladığınızda karşınıza bir GitHub sayfası çıkacak. Burada birden fazla paket bulunuyor. Windows işletim sisteminin desteklediği tür olan **.exe** uzantılı dosyayı indireceksiniz.

PHP'nin Kurulumu

Burada MAMP programını kullanacağız. Bu araç bizim için hem MySQL sunucusunu hem de PHP'yi kuruyor.

https://www.mamp.info/en/windows/

Programın ücretli sürümü de var; ama kullanmak zorunda değilsiniz. Şimdilik ücretsiz sürüm sizin için yeterli olacaktır.

Composer Kurulumu

PHP'nin kendisini taşıdığı bir topluluk bulunmaktadır. Bu topluktaki bazı kişiler herkesin işine yarayabilecek paketler çıkarmaktadır.

Bu paketleri kurabilmemiz için Composer'a ihtiyacımız bulunuyor.

https://getcomposer.org/doc/00-intro.md#installation-windows

Burada bulunan **Composer-Setup.exe** adında paket var. Bu paketi kurduktan sonrasında composer paket yöneticisini başarılı bir şekilde kurmuş olacaksınız.

Daha sonrasında terminali açıp **composer** yazarak kullanabileceğiniz komutları görürsünüz. Burada gördüğünüz `search` komutu sayesinde kullanabileceğiniz bütün paketleri arayabilirsiniz. Kendisi bize laravel projemizi oluştururken yardımcı olacak.

macOS için Kurulumlar

Editör Kurulumu

Windowsta olan editör kurulumları ile aynı çalışıyor. Kurulumlar ve kullandığımız araçların hepsi aynıdır.

Terminal Kurulumu

Windows kurulumlarında harici bir terminal indiriyorduk. Ama macOS işletim sisteminin kendi içerisinde bir terminal geliyor. Daha iyi bir terminal istiyorsanız **iTerm2** terminalini kurabilirsiniz. Web adresi: ***https://iterm2.com/***

Sayfaya girdiğinizde karşınıza **Download** tuşu çıkacak. Tıkladığınızda paket inmeye başlayacak.

2

Temeller

Bu Bölümde

Bu bölümde, Laravel'e yeni başlayan kişilerin ilk olarak bilmesi gereken temeller bulunuyor. Bölüm sonunda Laravel hakkında temel seviyede bilgi sahibi olacaksınız.

Route Yapisinda View'lar Nasil Yüklenir?

Kurulumlardan sonra Laravel'in temellerine giriş yapalım. İlk bölümde **Route** yapısını işleyeceğiz.

Laravel'de projenizi kurduktan sonrasında çok sayıda dosya ile karşılaşıyorsunuz. Ve ilk görüşte bunların fazlalığından dolayı gözünüz korkabilir. Böyle bir düşünceye kapılmanızı istemem. Bu kadar dosyayı görüp incelerken **routes** adında bir dosya göreceksiniz. Bu dosya altında birkaç tane PHP belgesi göreceksiniz. Bu dosyalardan şu anlık bir tanesi bizi ilgilendiriyor.

Projemizin dosya yolu: **routes/web.php**

Belge içerisinde bir `get` işlemi görüyorsunuz. Burada ilk parametre olarak sizden bir yol istiyor. Burada istediğiniz herhangi bir yol verebilirsiniz. Dilerseniz bir tane route yapısı üzerinden ilerleyelim. Siz **web.php** dosyasını açtığınızda karşınıza bir yapı çıkıyor.

Yapıda karşınıza çıkan temel bir routing yapısı. Bu yapıyı biraz anlayalım.

Yapıda eğer ki gönderilen değer **/** dizini ise bir fonksiyon çalıştırıyor. Ve bu fonksiyonun geri döndürdüğü değer bir view işlemi gerçekleştiriyor. Burada bulunan view işlemi bir `welcome` değeri çağırıyor, bu `welcome` değeri:

resources/views/welcome.blade.php bu yoldan geliyor.

Gönderilen bu değer fonksiyonda işlenip kullanıcıya gösteriliyor. Welcome view'nın yanında: **.blade.php** uzantısı var.

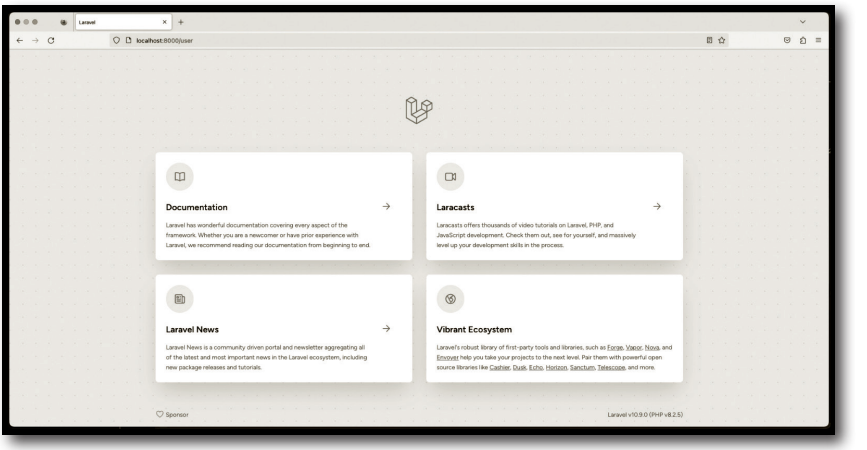
Bu uzantıyı döndürülen değerin devamına yazmanıza gerek yok.

Route yapısında gönderilen link yapısını değiştirip bir test edelim. Örnek olarak kullanıcıların oturum açtığında karşılaştırıldığı sayfaya yönlendirme işlemi yapalım.

Bunun için ilk değer olan yolu `'/user'`a çevirmemiz lazım. Daha sonrasında sayfaya baktığınızda karşınıza bir hata çıkacak. Şu anda anasayfada bulunduğumuz için bu hata ile karşılaşıyoruz. Bu hatanın kendisi Laravel'in içinde bulunan `abort` fonksiyonu sayesinde karşınıza çıkıyor.

Belirttiğimiz dizine gitmemiz için link yapısında değişiklik yapmamız gerekiyor. Link yapısının: **localhost:8000** olması gerekiyor.

Laravel projemizi başlattığımızda bu link üzerinden ilerlemiştik. User yönlendirmesi yapması için link yapısını: **localhost:8000/user** olarak çevirmemiz gerekiyor. Sonuç olarak alacağımız görüntü aşağıdaki gibidir.



Böyle bir çıktı ile karşılaşıyorsunuz. Projemiz halen çalışmaya devam ediyor.

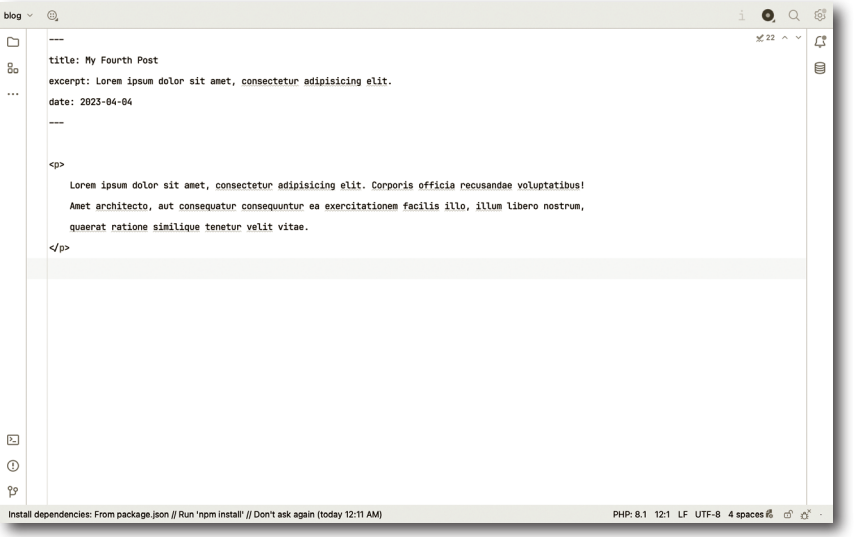
Şimdi ise View tarafında ufak bir düzenleme yapalım.

resources/views/welcome.blade.php bu dizine gelelim ve burada Laracasts'ın kendi yazısını **strong** etiketini kullanarak değiştirelim. Sayfayı yenilediğimizde bu yazının kalın bir biçimde yazıldığını görüyorsunuz.

Route sisteminde temel işlevler bu kadar. Daha sonraki bölümlerde api işlemleri için farklı bir route belgesi ile ilgileneceğiz.

Eklediğimiz metadatanın son hali aşağıda bulunmaktadır.

HTML Dosyasının Yolu: **resources/posts/my-fourth-post.html**



METADATALAR İÇİN COMPOSER PAKETİNİ KURMA

Bir önceki kısmın sonlarında sayfamıza metadata eklemiştik. Yapmamız gereken şey paket yöneticisi olan composer'ı kullanarak topluluk ile yapılmış **yaml-front-matter** paketi ile çalışmak.

Paketin kendisi yazdığımız metadataları işliyor. Daha sonrasında biz `Post` sınıfımızı kullanarak içerisine bir metot yazacağız. Bu metot sayesinde aldığımız değerlerin kendisini kullanarak dinamik bir sistem oluşturacağız.

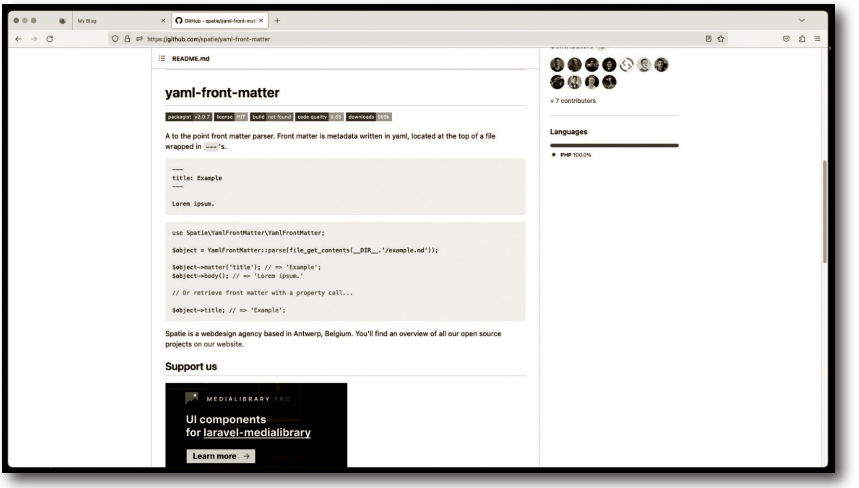
Bu işlemlerden önce paketi kurmamız gerekiyor. Kendisi **github.com** üzerinde açık kaynak halinde yayınlanıyor.

Yapmamız gereken şey:

https://github.com/spatie/yaml-front-matter linkine gitmemiz.

Sayfanın aşağılarında kurulumu ve bu paketin nasıl kullanıldığını anlatıyor.

Bir sonraki sayfda yer alan görseli inceleyebilirsiniz.



Yapmamız gereken şey metadata'nın bulunduğu belgeyi parse ediyoruz. Daha sonrasında **body** ve **matter** işlevlerini kullanarak istediğimiz herhangi bir datayı ya da bütün içeriği alabiliyoruz.

Paketi kurmamız için ilk bölümde kurulumunu yaptığımız composer paketini indirmiş olmanız lazım.

```
composer require spatie/yaml-front-matter
```

Paketi projemizin içerisine kurduktan sonra kullanıma hazır hale gelecektir.

İlk olarak **routes/web.php** belgesine gidiyoruz. İlk route'ımızın için bulunan işlemi yorum satırı içerisine alarak durduruyoruz. Paketimizin içeriğini kullanmaya başlayalım.

PHP Dosyasının Yolu: **routes/web.php**

```
$document = YamlFrontMatter::parseFile(
    resource_path("posts/my-fourth-post.html")
);
```

Yazdığımız kodda **my-fourth-post.html** dosyasını parse ediyoruz ve bunu bir bir değişkene aktarıyoruz.

3

Blade

Bu Bölümde

Bu bölümde Laravel'in görsel motor olarak kullandığı Blade'i temellerini göreceksiniz. Temel direktiflerin ne olduğunu ve nasıl kullanıldığını öğreneceksiniz.

BLADE'İN TEMELLERİ

Views dizinimizin altında bulunan belgelerde **.php** uzantısı dışında bir de **.blade** uzantısının olduğunu göreceksiniz. Blade bir görsel motorudur. PHP yazımını ve bazı tamplate işlemlerini çok kolaylaştırır.

Temellerimizi **posts.blade.php** üzerinden öğrenelim.

```
<?= $post->title ?>
```

Bu kodda title'ı ekrana yazdırıyoruz. Bunu bu şekilde yazmamıza gerek yok. Blade'i kullanarak yazımı aşağıdadır.

```
{{ $post->title }}
```

Şu anda çalışıyor ama **.blade** uzantısını kaldırıp belgenin ismini sadece **posts.php** yaparsak çıktı olarak dinamik bir şekilde linkleri almak yerine sadece yazıyı gösterecektir.

Peki, bu işlemleri blade'in kendisi nerede bulunuyor? Bunu bilmemiz gerekiyor. Bu işlemlerin **log** olarak kaydedildiği belge: **storage/freamwork/views** burada bulunuyor.

Bu dizine eriştiğinizde ismi anlamsız olan dosyalar karşınıza çıkacaktır. Bu dosyaların ismi herkese göre farklı değerler olduğu için sizin bulmanız gerekiyor.

Doğru belgeyi bulmanız için **views** dosyasına yazdığınız kodun blade anlamında eş değer koda sahip olan belge doğru belge olacaktır.

Buna örnek verelim:

```
<h1>
    <a href="/posts/<?= $post->slug ?>">
        {{ $post->title }}
    </a>
</h1>

<div>
    <?= $post->excerpt ?>
</div>
```

Böyle bir kodun log tutulan bölümü aşağıdaki gibi olmalıdır.

```
<h1>
    <a href="/posts/<?= $post->slug ?>">
            <?= $post->title ?>
    </a>
</h1>

<div>
    <?= $post->excerpt ?>
</div>
```

Siz her ne değişiklik yaparsanız o da kaydedilecektir.

Sırada **post.blade.php** belgemize gidelim. İçeriğe baktığımızda `body()` işlevini kullanmışız. Bunu eğer ki {{ }} ile yaparsak bize olduğu gibi gösterecektir.

Bu tip işlevler için {!! !!} bunu kullanırız. İçerisine değerimizi verebiliriz.

Örnek Kullanım

```
{!! $post->body !!}
```

Başarılı bir şekilde çalışıyor.

Son olarak title işlemini de düzenleyip bu belgeyi kapatabiliriz.

Düzenledikten sonrasında:

PHP Dosyasının Yolu: **post.blade.php**

```
<article>
    <h1>{{ $post->title }}</h1>

    <div>
        {!! $post->body !!}
    </div>
</article>
```

Sonrasında aynı düzeltmelerin kendisini diğer view'ımız içinde yapalım.

PHP Dosyasının Yolu: **posts.blade.php**

```
<?php foreach ($posts as $post) : ?>
    <article>
        <h1>
            <a href="/posts/{{ $post->slug }}">
                {{ $post->title }}
            </a>
        </h1>

        <div>
            {{ $post->excerpt }}
        </div>
    </article>
<?php endforeach; ?>
```

Şu anda posts view'ımız içerisinde `foreach` işlemi yapıyoruz. Ama okunuş açısından biraz daha iyileştirilmeler yapılabilir. Blade'in kendisi bize burada direktifleri sunuyor.

Normal bir foreach işlemi:

```
foreach($tests as $test) {

}
```

Blade motoru üzerinde yazılan foreach işlemi:

```
@foreach ($tests as $test)

@endforeach
```

Bu sayede yukarıda da belirttiğim gibi okunuş tarafında daha güzel bir hal alıyor.

Bunu view'ımıza entegre ettiğimizde aşağıdaki çıktıyı göreceksiniz.

```
@foreach($posts as $post)
    <article>
        <h1>
            <a href="/posts/{{ $post->slug }}">
```

4

Veritabanları ile Çalışma

Bu Bölümde

Bu bölümde, veritabanlarını nasıl yazdığımız projeye entegre edip veri ekleme, silme, düzeltme işlemlerini nasıl yapabiliriz bunları öğreneceğiz.

ORTAM DOSYALARI VE KURULUMLAR

Veritabanı üzerinden veri çekme, ekleme ve veri silme işlemlerini yapmadan önce kullandığımız Laravel Freamwork'ünde veritabanı bağlantısı nasıl yapılır bunu öğrenmemiz lazım.

Bunun için proje klasörümüzde bulunan **.env** belgesini kullanıcağız. Belge içerisinde veritabanı işlemleri dışında mail gönderme AWS (Amazon Web Services) işlemleri yapılabiliyor. Bizim yapmamız gereken **DB** bölümünü bulmak olacak.

Baktığımızda veritabanı için hangi sistemi kullanacağız, **host** ve **port** numaralarımız gibi içerikleri bizden istiyor. Aslında bizden saf PHP yazarken klasik olarak istenen bilgileri talep ediyor. Burada bulunan bilgileri kendinize göre kullandığınız hosta göre şekillendirebilirsiniz.

Benim kullandığım bilgiler aşağıdaki gibidir;

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

NOT Burada bulunan `DB_DATABASE = blog` bölümü kullanmak istediğiniz veritabanın ismidir. Burayı değiştirdiğinizde unutmamanız gereken şey kullandığınız ismi veritabanını oluştururken kullandığınız isimle aynı olması gerekiyor.

Sırada veritabanını oluşturmamız lazım. Bunun için MySQL'i kullanacağız. Kitabın başlarında kurulumları yapmıştık. Eğer ki geri dönüp bakar iseniz her iki işletim sistemi içerisinde MySQL kurulumunu yaptık.

Sırada MySQL'e erişmemiz gerekiyor. Bunun için terminale aşağıdaki komutu yazacağız.

```
mysql -uroot -p
```

komutunu yazdığımızda bizden şifre istediğiniz görüyoruz. Eğer ki daha öncesinde MySQL'de herhangi bir işlem yaptıysanız oraya kendi kullandığınız şifreyi yazabilirsiniz; ama sıfırdan bir kurulum yaptıysanız herhangi bir şifre bulunmayacaktır. Bunun için sadece **Enter** tuşuna basmanız yeterlidir.

Bir hata almadıysanız karşınıza MySQL monitörü çıkacaktır. Artık MySQL komutlarımızı yazabiliriz.

Aşağıda karşımıza çıkan MySQL monitörü bulunuyor.

Artık veritabanımızı oluşturduğumuza göre **migration** işlemleri için gerekli tabloları veritabanımız içerisine gönderebiliriz.

İlk olarak monitörden çıkmamız gerekiyor. Bunun için terminale:

```
exit
```

yazmamız yeterli olacaktır.

Şimdi bütün migrate'in içerisinde bulunan tabloları oluşturduğumuz veritabanına göndermemiz gerekiyor. Veritabanına göndermek için:

```
php artisan migrate
```

komutunu kullanacağız. Eğer ki hata almadıysanız başarılı bir şekilde gönderme işlemimiz yapılmış demektir.

Şimdi test edelim. MySQL monitörümüze tekrar bağlanmamız gerekiyor. Daha sonrasında veritabanımız içerisine girmeliyiz. Aşağıda bulunan sorguyu kullanabiliriz.

```
use blog;
```

Veritabanımız içerisine girdiğimize göre sırada yapmamız gereken şey içerisinde bulunan tabloları görmemiz gerekir.

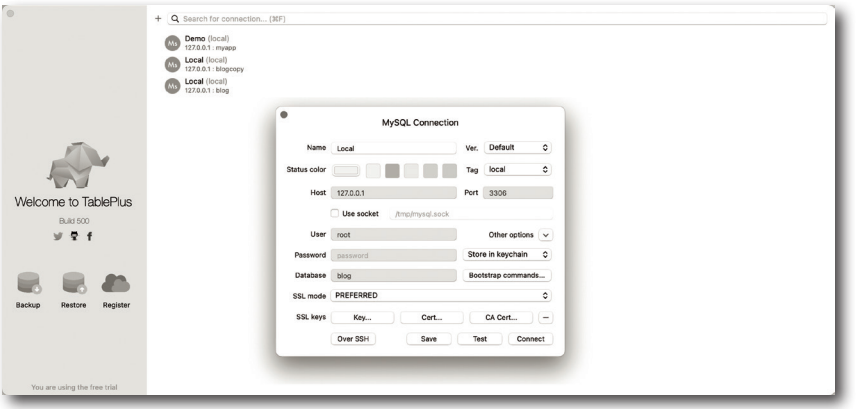
Aşağıdaki komutu kullanabiliriz.

```
show tables;
```

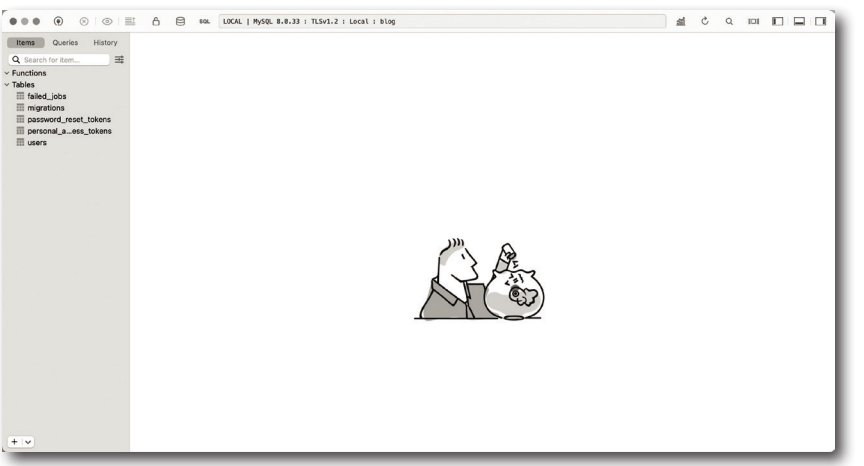
Eğer ki içerisinde birden fazla tablo görüyorsanız başarılısınız. Peki, biz bu veritabanına nasıl erişebiliriz. Yani kendisini nasıl daha detaylı bir şekilde yönetebiliriz?

Bunun için kurulumlarda indirdiğimiz **Table Plus** programını kullanacağız. Kendisini belli kısıtlamaları kabul ederek kullanabilirsiniz. Her ücretli programın alternatifi olduğu gibi kendisinin de alternatifleri bulunmaktadır.

Bağlantı için ilk olarak Table Plus programımıza giriş yapıyoruz. Daha sonrasında sağ tık yapıp yeni bağlantı diyerek yeni bir işlem oluşturuyoruz. Artık sizden veritabanı bağlantısının yapıldığı gibi birkaç bilgi istiyor. Bunları doldurmamız lazım. Aşağıda benim için geçerli olan bilgileri göreceksiniz.



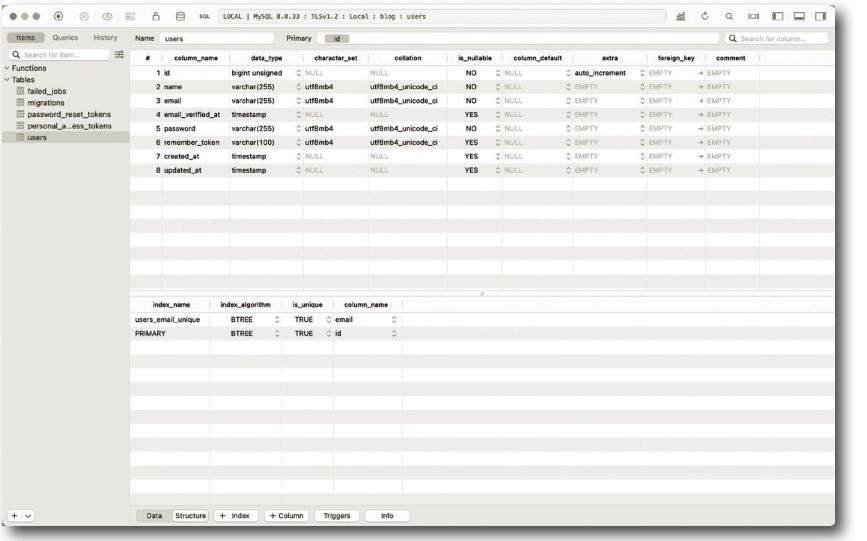
Connect butonuna tıkladığınızda aşağıdaki sayfa karşınıza çıkacaktır.



MIGRATIONS: TEMELLER

Veritabanı bağlantı işlemleri bittiğine göre sırada Migrations bölümüne geçebiliriz. Migration sayesinde veritabanı kontrolünüz çok kolay bir hal alacaktır. O zaman öğrenemeye başlayabiliriz.

İlk olarak **Table Plus** programımızdan **users** tablomuza giriş yapıyoruz. Devamında aşağıda bulunan **Structure** butonuna tıklamamız gerekiyor ki yapıyı daha detaylı görebilelim.



Peki, bu tabloların içerisinde bulunan veriler havadan mı geliyor? Kesinlikle HAYIR. Bu işlemlerin yapıldığı yeri bulmak bulmak için aşağıda bulunan yola gitmemiz lazım.

`blog/database/migrations/2014_10_12_000000_create_users_table.php`

Burada `Schema` adında bir sınıf göreceksiniz. Kendisi içerisinde bulunan `create` metodu sayesinde yeni bir şema oluşturuyor.

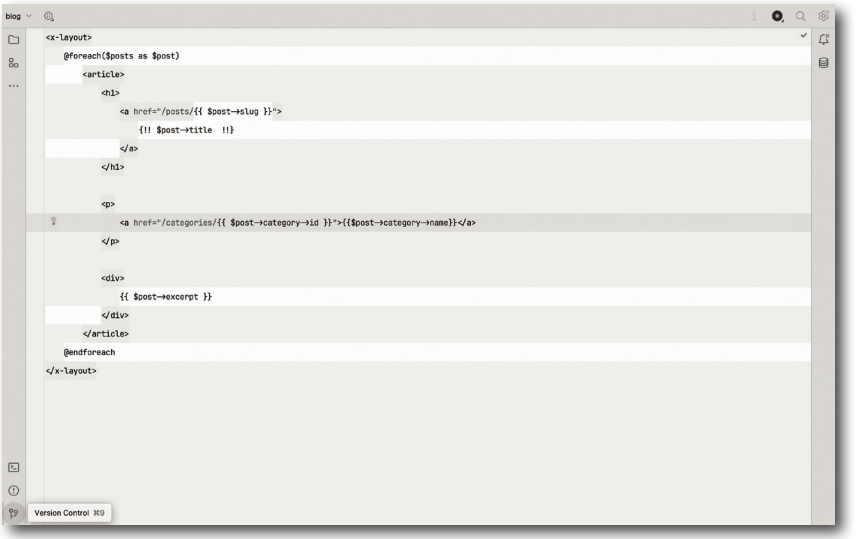
Metodun kendisi ilk olarak oluşturulacak tablonun ismini birinci parametre olarak alıyor. Daha sonrasında ekleyeceği verileri bir fonksiyon içerisinde işliyor. İlk olarak fonksiyonun içerisine bakalım.

İlk olarak **posts.blade.php** belgemize gidelim. Daha sonrasında kullandığımız Category değerinin id'sini bütün Post'larımıza entegre edelim.

Belgeye gittiğimizde **category** kısmını yazdırdığımız kısımda link bölümünü **diyez #** işareti ile doldurmuştuk. Artık elimizde bir yapı bulunuyor. Bu yapının id değerini alarak link oluşturabiliriz. Aşağıda olması gereken hali gösteren bir görsel bulunuyor.

PHP Dosyasının Yolu

`resources/views/posts.blade.php`



Aynı kodu **post.blade.php** belgemizde bulunan etiketimizin içerisine yapıştırıyoruz. Sayfaya baktığımızda artık Post'larımızın herhangi birisine tıkladığımızda **category** numarasına göre başarılı bir şekilde gruplandırabiliyoruz.

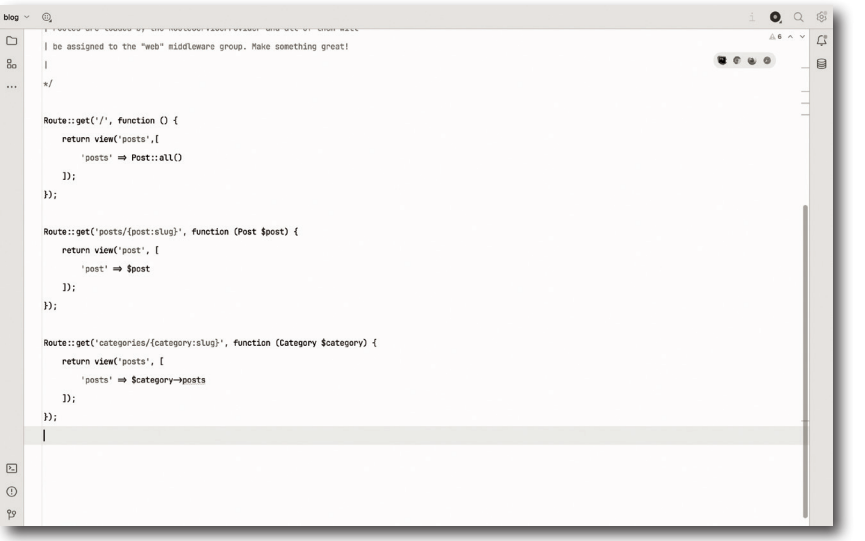
Peki, biz category numarası yerine `slug` değerleri ile bu işi tamamlarsak daha iyi bir sonuç alabiliriz.

Bunun için hem **post.blade.php** belgemizde hem de **posts.blade.php** belgemizde bulunan `id` değeri yerine `slug` değerini yazar isek artık `slug matter`'ı üzerinden veri gösterebiliriz. Fakat şu anda herhangi bir Post'a tıkladığınız da bir yere varamayacağınızı görürsünüz. Bunun sebebi biz route yapımızda sadece `category` değerini kullandık eğer ki kendisinin yanına `slug` değerini ekler isek başarılı bir şekilde Post'larımızı alabiliyoruz.

Aşağıda route yapımızın görseli bulunuyor.

PHP Dosyasının Yolu

`routes/web.php`



CLOCKWORK VE N + 1 PROBLEMİ

Bir önceki bölümde **N + 1** problemini ortaya çıkardık. Bu problemin kendisi her bir postu eklediğimizde yeni bir SQL sorgusu oluşturmaktadır. Buda belli bir süre sonrasında projenizin bulunduğu sunucuda yetersiz alandan dolayı çökmelere veya yavaşlamalara sebep olabiliyor.

İlk olarak bu problemin ortaya çıkardığı tekrarlama işlemini log olarak alacağız.

Bunun için ilk Route'ımızda veritabanımızı dinledikten sonrasında kendisinin içerisinde bulunan sorguyu `logger()` fonksiyonumuzu kullanarak döndürülen bütün değerleri bir sayfada tutacağız.

Bunun için aşağıda bulunan kodu kullanabiliriz.

```
\Illuminate\Support\Facades\DB::listen(function ($query) {
    logger($query->sql)
});
```

Sayfaya dönüp yeniliyoruz.

Şu anda **log** kaydetme işlemimiz yapıldı. Peki, nereye yaptı! Kaydedilen log'ları görmek için **storage/logs/laravel.log** belgesine gittiğimizde karşımıza proje başladıktan sonrasında tutulan bütün log'lar çıkıyor. Fakat sizde sorgu dışında dosya yolları vb. bilgilerde çıkıyor olabilir. Bizim işimiz sorgular ile olduğu için log belgemizin en aşağı kısmına gittiğimizde sql sorgularını görüyoruz. Aşağıda proje her yenilendiğinde tekrar çalıştırılan sorgu bulunmaktadır.

```
[2023-07-10 12:07:34] local.DEBUG: select * from `posts`

[2023-07-10 12:07:34] local.DEBUG: select * from `categories` where `categories`.`id` = ? limit 1

[2023-07-10 12:07:34] local.DEBUG: select * from `categories` where `categories`.`id` = ? limit 1

[2023-07-10 12:07:34] local.DEBUG: select * from `categories` where `categories`.`id` = ? limit 1
```

Peki, SQL kısmının log'unu nısl olabildik? Bunun sebebi oluşturduğumuz işlevin içerisinde bulunan `$query` parametresini `logger()` fonksiyonu içerisinde `$query->sql` olarak kullandığımız içindir. Bir önceki derslerde bağdaştırıcı kullanmıştık. O zaman hem sorgu hemde bağdaştırıcıyı bize vermesi için `logger` fonksiyonumuzun içerisine `$query->bindings` parametresini eklememiz yeterli olacaktır. Aşağıda fonksiyonun son hali bulunmaktadır.

```
logger($query->sql, $query->bindings);
```

Bunu göstermemin sebebi istediğiniz veriyi log'layabilirsiniz.

Peki, şu anda bir problemimiz bulunuyor. Başta söylediğim performans kaybı problemini nasıl kapatabiliriz. Yani daha kısa sürede sorgu biriktirmeden nasıl kısa sürede daha fazla veri geçişi sağlayıp performansımızı arttırabiliriz.

Bunun için topluluğun gücünü kullanabiliriz. Yani bazı kişilerin paylaştığı paketleri kullanabiliriz. Biz burada **Clockwork** paketini kullanacağız. İlk olarak ilk başta yazdığımız SQL log'larını yazdıran kodu siliyoruz. Paketi kurmak için aşağıdaki komutu çalıştırıyoruz.

```
composer require itsgoingd/clockwork
```

Paketi kurduktan sonrasında kendisini eklenti olarak da kurmamız gerekiyor. Eklenti sadece **Chrome** ve **Firefox** tarayıcılarını destekliyor.

5

Tasarımı Entegre Etme

Bu Bölümde

Bu bölümde, yazdığımız blog sayfasının daha hoş ve güzel olması için GitHub'da bulduğumuz ünlü bir sayfanın temasını entegre edeceğiz ve ufak ayarlamalar yapacağız.

HTML VE CSS'İ BLADE'E DÖNÜŞTÜRME

İlk olarak temanın bulunduğu sayfaya gidelim. Ben burada tasarım açısında kitapta olmayan bir konu olan tema hazırlamak yerine nasıl hazır bir temayı sayfaya entegre edebilirsiniz bunu göstermek istiyorum. Bu konuda popüler ve herkese açık olan kaynakları kullanmakta fayda görüyorum.

İlk olarak:

https://github.com/laracasts/Laravel-From-Scratch-HTML-CSS linkine gidelim.

GitHub sayfasında aşağılara doğru kaydırırsanız karşınıza temanın nasıl olduğunu göreceksiniz.

> **NOT** Eğerki tasarımda hoşunuza gitmeyen bir kısım yada Logo'da bir problem görüyorasanız kendinize ait aynı boyurlarda bir Logo oluşturup eski Logo'yu silip kendisi yerine koyabilirsiniz. Burada sildiğiniz Logo ile oluşturduğunuz Logo'nun aynı olmasına dikkat edin.

İlk olarak **Code** yazan **Yeşil** butuna tıklamanız gerekiyor. Burada size indirmek için birden fazla seçenek sunuyor. Fakat biz burada: **Download ZİP** indirme seçeneğini kullanacağız.

Dosyayı açtığımızda karşımıza görsellerin bulunduğu **images** klasörü sayfanın tamamının bulunduğu **index.html** ve bir post'a tıkladığınızda karşınıza çıkan **post.html** sayfası olmak üzere üç adet parçayı görüyoruz. Artık bu tasarımı sayfamıza yerleştirmemiz gerekiyor. İlk olarak **dosyalar/finder** ya da herhangi bir klasör yöneticisiniz açıyorsunuz. İndirdiğimiz tasarımda bulunan **images** klasörünü kopyalayıp projemizin **public** dizininin altına yerleştiriyoruz.

Devamında **index.html** belgemizi kullandığınız **IDE/Editör**'de açıyoruz. İçerisin de bulunan kodların hepsini kopyaladıktan sonrasında projemizde bulunan:

resources/views/components/layout.blade.php

belgemize gidip bütün içeriğini sildikten sonrasında kopyaladığımız kodu yapıştırıyoruz.

Sayfayı yenilediğimiz de temanın geldiğini görüyoruz fakat görsellerin sadece alt başlıkları bulunuyor. Yani klasör yolu kısmında bir problemimiz bulunuyor. Bu problemi bütün **img** etiketlerini düzenleyerek çözebiliriz.

İlk olarak **CMD/Command + F** kısayolunu **IDE/Editor**'de kullanıyoruz. Devamında bir arama kutusu yukarıdan geliyor, içerisine `img` yazarak bütün etiketleri bulabiliyoruz. Burada hepsinin düzenlenmesi gereken konu yol belirtilirken hepsinin başına nokta koyulmuş. Bulunan noktaları silersek bütün görsellerin geldiğini göreceksiniz. Aşağıda bir örneği bulunuyor.

Önceki hali;

```
<img src="./images/logo.svg"
```

Çalışan hali;

```
<img src="/images/logo.svg"
```

Artık sayfamızı yenilediğimizde başarılı bir şekilde bütün görsellerin geldiğini görebiliyoruz.

Sırada temayı Blade'e çevirmemiz gerekiyor. İlk olarak yirmi altıncı satır ile dört yüz altmış sekizinci satır arasını kesiyoruz. Şu anda sayfamızda sadece `<nav></nav>` ve `<footer></footer>`'ın olması gerekiyor. Bu ikisi arasına Blade bölümünde görüdüğümüz `$slot` değerini yerleştiriyoruz.

```
{{ $slot }}
```

Devamında **posts.blade.php** belgemize gidelim. Burada `<x-layout></x-layout>`'u kullanmıştık. İçerisinde bulunan bütün kodları tekrar kullanma ihtimaline karşı yorum satırına alalım. Ve aynı şekilde x-layout'umuzun içerisine kopyaladığımız kodlarımızı yapıştıralım. Sayfamıza baktığımızda `$slot` işlemimizin başarılı bir şekilde çalıştığını görüyoruz. Şu anda link kısmında muhtemelen bir category içerisindesiniz. Linki ilk haline getirelim.

```
localhost:8000
```

Sayfada gördüğünüz her bir **card** aslında bir **article** etiketi içerisinde bulunuyor. Şöyle bir şey yapalım, article etiketlerimizden bir tanesini örnek olarak class değeri `lg:grid lg:grid-cols-3` olan div'in içerisinde ki `article` etiketlerinden en başındaki keselim.

Devamında `components` klasörümüzde bu card'larımızı kullanacağımız yeni bir blade belgesi oluşturalım.

`post-card.blade.php`

ve kopyaladığımız article etiketini içerisine yapıştıralım. Devamında aynı div'in içerisinde kalan `article` etiketlerini silebiliriz.

Sayamızı yenilediğimizde sildiğimiz card'ın olmadığını görüyoruz. Fakat şu anda kendisi bir component yani kendisini aynı div'in içerisine `<x-post-card />` yazarak çağırabiliriz. Bu işlemi yaptıktan sonrasında sayfaya baktığımızda bir tane olduğunu görüyoruz. Aynısından iki tane daha ekleyebiliriz. Üst tarafta bir adet daha div olduğunu görebilirsiniz ve aynı şekilde içerisinde article etiketleri bulunuyor. Bunların hepsini temizleyip aynı koddan iki tane de oraya yazabiliriz. Sayfaya baktığımızda herhangi bir değişikliğe uğramadığını görüyoruz.

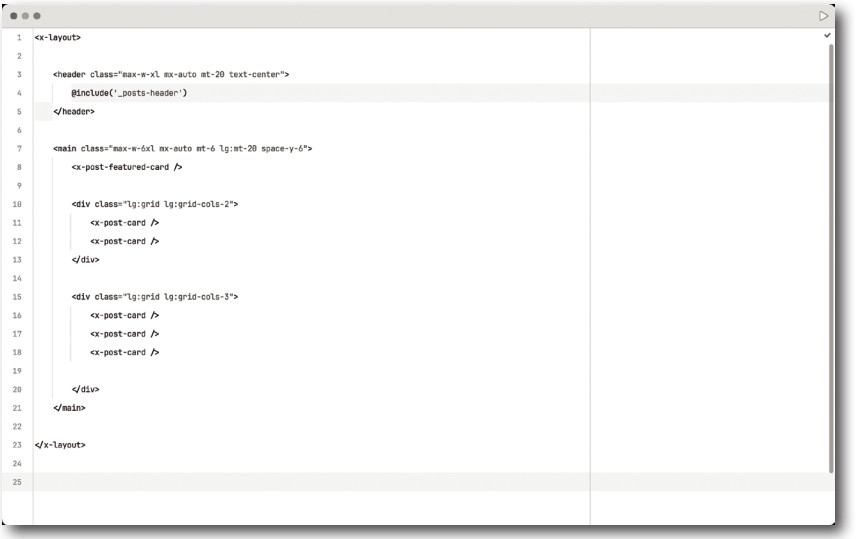
Fakat biz bu işlemin aynısı en üstteki article etiketini silip yapamıyoruz. Bunun sebebi kendisinin CSS kodu diğer post'larımızdaki gibi sadece **div** etiketinde değil kendi içerisinde de bulunuyor. Bunu şöyle halledebiliriz. Article etiketini keselim ve kestiğimiz kodu bir `component` olarak kullanalım. Bu yüzden **component** klasörüne yeni bir adet belge daha ekleyeceğiz.

`post-featured-card.blade.php`

devamında kestiğimiz article etiketini içerisine yapıştıralım. Artık kestiğimiz kısma `<x-post-featured-card />` yazarak componentimizi kullanabiliriz. Şu anlık sayfamızda bir problem bulunmuyor. Son olarak `header` kısmı kaldı bunu bir component olarak kullanmak saçma olur çünkü sadece bir değeri temsil etmiyor. Aslında cümle biraz değişik gelmiş olabilir; ama kendisini anlık olarak değil daha sonraki bölümlerde işimizi kolaylaştırsın diye birden fazla kez kullanıcağız. Bu yüzden kendisini `@include` ile kullanmak istiyorum.

İlk olarak kendisi içerisinde ki bütün değerleri keselim. Ve **views** klasörünün altına **_posts-header.blade.php** belgemizi oluşturalım. Devamında kestiğimiz kodları içerisine yapıştıralım. Bahsettiğim gibi kendisini **@include** ile kullanmak istediğim için kestiğimiz kısma `@include ('_posts-header')` olarak kodumuzu kullanırsak başarılı bir şekilde sayfayı çalıştırmaya devam ederiz. Belgemizin son hali aşağıda bulunuyor.

PHP Dosyasının Yolu: **resources/_posts-header.blade.php**



```
<x-layout>

    <header class="max-w-xl mx-auto mt-20 text-center">
        @include('_posts-header')
    </header>

    <main class="max-w-6xl mx-auto mt-6 lg:mt-20 space-y-6">
        <x-post-featured-card />

        <div class="lg:grid lg:grid-cols-2">
            <x-post-card />
            <x-post-card />
        </div>

        <div class="lg:grid lg:grid-cols-3">
            <x-post-card />
            <x-post-card />
            <x-post-card />

        </div>
    </main>

</x-layout>
```

Blade Bileşenleri ve CSS Grids

İlk olarak **posts.blade.php** belgemizde bulunan class'ı:

```
lg:grid lg:grid-cols-2 ve lg:grid lg:grid-cols-3
```

olan class'ları, div'leri yorum satırı içerisine alalım.

> **NOT** Blade'de yorum satırı {{--}} bu şekilde yapılıyor.

Devamında `<x-post-featured-card />` componentimize `:post="$post"` değerini ekleyelim. Burada bulunan **iki nokta** olmasa `$post` değeri bir değişken olarak değil bir metinsel ifade olarak algılanıyor.

Sırada **post-featured-card.blade.php** belgemize gidelim. Article etiketimizin en üstüne post props'umuzu ekleyeceğiz.

```
@props(['post']);
```

Eklediğimiz bu değeri kullanalım. İlk olarak mt-4 class'ına sahip olan div'in içerisinde class'ı text-3xl olan h1 etiketinin içeriğini `$post->title` olarak değiştirelim.

6

Arama

Bu Bölümde

Bu bölümde, sayfamızda bulunan **Bir Şeyler Ara** kısmını nasıl çalıştırabiliriz.

İki farklı yöntem ile bunu öğreneceğiz.

ARAMA (ÇİRKİN YOL)

Sayfamızda Bir Şeyler Ara içeriğine sahip category'lerimizin yanında bulunan bir input var. Kendisini sayfamızda bulunan Post'larımızda arama yapmak ve aradığını bulmak isteyen kullanıcılarımız için aktifleştirmemiz gerekiyor. İlk olarak kendisinin içerisine `testing` yazıp **Enter** tuşumuza basalım. Şu anda sayfamızda beklediğimiz gibi bir değişiklik olmadı. Fakat link yapımıza baktığımızda **search** adında bir değerin gönderdiğimiz değeri içerisinde tuttuğu bir `GET` parametresi görüyoruz. İlk olarak bu değeri kullanalım. Bunun için **routes/web.php** belgemizi açalım. Burada anasayfa yani ilk Route'ımız bulunuyor. Ekleme işlemimizi bu Route'ta yapacağız. Route'ımızın içinde bulunan return işleminin üstüne biraz boşluk bırakalım. Yapmak istediğimiz şey bir koşulumuz olsun ve kendisi search isteğini kontrol etsin eğer ki bir değer gönderildi ise veritabanından gönderilen değer ile `title` veya `body` değerleri eşleşen Post'ları çekmesini istiyoruz.

Fakat bunu yapabilmemiz için `return` ifadesinin içerisinden kullandığımız `posts` değerinin içeriğini bir değişken olarak almamız gerekiyor. Bunun için içeriği bulunduğu yerden kesiyoruz ve açtığımız boşluğun içerisinde `$posts` adında yani bir değişken oluşturarak kestiğimiz değeri bu değişkene aktaralım. Değerin sonunda bulunan `get()` değerini silelim. Bunun sebebi zaten istek gönderiyoruz.

Devamında kesitiğimiz kısma geri dönüp oluşturduğumuz değişkeni `posts` değerimize aktaralım. Burada en sonunan `get()` işlevini kullanabiliriz. Aşağıda yaptığımız işlemlerin son hali bulunuyor.

PHP Dosyasının Yolu: **routes/web.php**

```php
/*
| Web Routes
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "web" middleware group. Make something great!
|
*/

Route::get( url: '/', function () {
    $posts = Post::latest();

    if (request( key: 'search')) {
        $posts
            ->where('title', 'like', '%' . request( key: 'search') . '%')
            ->orWhere('body', 'like', '%' . request( key: 'search') . '%');
    }
    return view( view: 'posts',[
        'posts' => $posts->get(),
        'categories' => Category::all()
    ]);
})->name( name: 'home');

Route::get( url: 'posts/{post:slug}', function (Post $post) {
    return view( view: 'post', [
        'post' => $post
```

Şu anda sayfamızın linkini eski haline çevirdiğimizde ve herhangi bir başlığın içerisinde bulunan bir kelimeyi arama kısmında arattığımız da başarılı bir şekilde karşımıza geldiğini görüyoruz.

Yazdığımız değerin **Enter** tuşuna bastıktan sonrasında halen orada kalmasını istiyoruz. Bunun için **_posts-header.blade.php** belgemize gidelim. Belgede aşağılara doğru kaydırdığımızda `Search` adında bir adet yorum satırı görüyorsunuz. Burası bizim search input'uzun kendisi ve içindeki değerlerden oluşuyor.

Yapmamız gereken şey `input` etiketimize gönderilen `search` değerini `value` olarak göndermek olacak.

```
value="{{ request('search') }}"
```

Artık sayfamıza baktığımız istediğimizin olduğunu görebiliyoruz.

Arama (Temiz Yol)

Bir önceki bölümde arama işlemini yapmanın çirkin halini görmüştük. Bu bölümde bu işi yapmanın temiz halini göreceğiz. İlk olarak terminalimizi açıp **PostController** adında yeni bir controller oluşturalım.

```
php artisan make:controller PostController
```

Devamında:

```
app/Http/Controllers/PostController.php
```

konumuna gidip controller'ımızı açalım. İçerisinde Controller sınıfından extends alınmış bir `PostController` adında bir sınıf bulunuyor. İçersine index adında bir adet metot oluşturalım.

```
public function index()
{
}
```

Bu metodumuz arama işleminin temellerini kapsayacak. İçerisine ekleyeceğimiz arama işlemini aslında bir önceki bölümde oluşturmuştuk.

Oluşturduğumuz işlev için: routes/web.php belgemize gidelim. Önceki bölümde işlem yaptığımız anasayfa Route'ımızın içeriğini bulunduğu yerden keselim. Kestiğimiz değeri oluşturduğumuz index metotumuzun içerisine yapıştıralım.

Şu anda bu işlevin bir anlamı bulunmuyor. Belki kullandığınız IDE'nin kendisi belki bu işlemi yapmıştır. Fakat yapmadıysa bunu belirtelim. Siz yapıştırma işlemi yaptığınız değer içerisinde bulunan sınıfları da controller içerisine dahil etmeniz gerekiyor. Aşağıda use ile sayfamıza dahil ettiğimiz sınıflar bulunuyor.

```
use App\Models\Post;
use App\Models\Category;
```

Devamında kestiğimiz route'ımızı düzenleyelim. Yapmak istediğimiz şey controller'ımızın altında daha demin oluşturduğumuz index metotunuz çalıştırmak. Aşağıda route'ımızın son hali bulunuyor.

```
Route::get('/', [\App\Http\Controllers\PostController::class, 'index'])->name('home');
```

Burada yol belirmek yerine belgenin üst tarafına use ile çağırdıktan sonrasında da Route::get('/', [PostController::class, 'index'])->name('home'); olarak da kullanabiliriz.

Şu anda sayfamıza baktığımızda proje'mizin halen çalıştığını görebiliyoruz.

Şu anda yaptığımız işlem aslında bakarsanız Route belgemizde bulunan ilk Route'ı daha temiz bir hale getirdi. O zaman aşağıda bulunan posts Route'ımız içinde aynısını yapabiliriz.

Aynı işlemi yapacağız. İlk olarak controller'ımıza geleceğiz ve içerisine index metotumuzun altına show adında yeni bir metot oluşturacağız. Kendisinin içerisine Post Route'ımızda kullandığımı Post $post değerini aynı şekilde metot'umuzun içerisine parametre olarak göndereceğiz.

Sonrasında **web.php** belgemizde bulunan post route'ımızın içeriğini aynı şekilde kesiyoruz. Kestiğimiz değeri controller'imizin içerisinde bulunan show metodumuzun içerisine yapıştıracağız ve içeriğini kestiğimiz route'ı aşağıdaki hale getiriyoruz.

```
Route::get('posts/{post:slug}', [PostController::class, 'show']);
```

Şu anda sayfamızda bulunan herhangi bir Post'a tıkladığımızda başarılı bir şekilde karşımıza geldiğini görebiliyoruz.

7

Filtreleme

Bu Bölümde

Bu bölümde, sayfamızda bulunan Post'lara belli filtrelemeler getirerek kullanıcıların isteklerine uygun en iyi Post'u göstermeyi hedefleyeceğiz.

SORGU KISITLAMALARI

Örnek olarak bir kullanıcı hem şu category'de olsun hem de içerisinde şu yazı bulunsun gibi bir istek gönderebilir.

Bunun için ilk olarak **PostController.php** belgemize gidelim. İçerisinde bulunan `index` metotunun içerisinde `filter()` işlevini kullanmıştık. İşlevin içerisinde bulunan `'search'` parametresinin yanına yeni bir parametre daha gönderelim. Hatırlarsanız bu değerlerin hepsini bir önceki ders bir dizi mantığında hepsini toplamıştık. Yanına başlangıçta verdiğim örnek yola çıkarak `'category'` parametresini gönderebiliriz.

Devamında sayfada bulunan en üstteki Post'un `category` değerini category menümüzden bulup tıklayalım. Şu anda link yapımıza baktığımızda `/categories/.....` gibi bir yapı görüyoruz. O zaman biz bu yapıyı çevirelim ve aşağıdaki hale getirelim. /?category=….. Değiştirdiğimizde bizi anasayfaya attığını görebiliyoruz. Bunun sebebi `search` kısmında **Post.php** belgemizde yaptığımız işlemi kendisine yapmamamız.

Bunun için ilk olarak **Post.php** belgemize gidiyoruz. Sonrasında `scopeFilter` işlevimizin içerisinde bulunan `when` işlevini kopyalıyoruz ve kendisinin aşağısına yapıştırıyoruz ve kendisini category'e uyarlıyoruz. Aşağıda son hali bulunuyor.

PHP Dosyasının Yolu: **app/Http/Models/Post.php**

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


class Post extends Model
{
    use HasFactory;

    protected $guarded = [];

    protected $with = ['category', 'author'];

    no usages
    public function scopeFilter($query) {
        $query->when($filters['search'] ?? false, fn($query, $search) =>
            $query
                ->where('title', 'like', '%' . $search . '%')
                ->orWhere('body', 'like', '%' . $search . '%'));

        $query->when($filters['category'] ?? false, fn($query, $category) =>
        $query
            ->where('title', 'like', '%' . $search . '%')
            ->orWhere('body', 'like', '%' . $search . '%'));
    }

    public function category()
    {
        return $this->belongsTo( related: Category::class);
    }
```

Fakat istediğimizi yapabilmemiz için bu yeterli değil. Bizim burada filterleme işlevi için kullandığımız whereHas adında bir işlevimiz bulunuyor. Kendisini kullanarak eğer ki category'lerimizin slug değerlerine göre bir filtreleme sistemi oluşturursak bizim için yeterli olacaktır.

O zaman scopeFilter işlevimizde bulunan category işlemini:

```
$query->when($filters['category'] ?? false, fn($query, $category) =>
    $query->whereHas('category', fn ($query) =>
       $query->where('slug', $category)
    )
);
```

bu hale çevirelim. Şu anda sayfamızı category get değeri olarak gönderilmiş halde yenilerseniz ve Clockwork aracını açıp **Database** kısmına bakarsanız istediğimiz sorgunun döndürüldüğünü görebilirsiniz.

Son olarak link yapısı ve seçili category'nin kalmasını sağlayan currentCategory işlevini düzenleyelim.

İlk olarak **routes/web.php** belgemize gelelim. Belgede bulunan categories route'ımızı yorum satırına alalım. Çünkü kendisinin yaptığı işlemi neredeyse yaptık sayılır.

Devamında **_posts-header.blade.php** belgemize gidelim. Belgede bulunan @foreach'imizin içerisinde bulunan x-drop-down component'imizin içerisine bulunan href kısmının değerini /categories/{{ $category->slug }} aşağıdaki hale çevirelim.

Son olarak **PostController.php** belgemizde bulunan index metotumuzun içerisinde ki return işlevine bir değer daha ekleyelim.

```
'currentCategory' => Category::firstWhere('slug', request('category'))
```

Şu anda **routes/web.php** belgemizde yorum satırına aldığımız category route'ımızı silebiliriz. Artık aşağıdaki gibi bir link yapısı yazdığınızda başarılı bir şekilde sonuç aldığınızı görebilirsiniz.

```
/?search=…..&category=…..
```

Blade Componentlerini Kategorileştirme

İlk olarak **_posts-header.blade.php** belgemize gidelim. Belgede önceki derslerde eklediğimiz x-dropdown component'imiz bulunuyor. Kendisini bulunduğu yerden keselim.

Kendisini bütün değerlerini `<x-category-dropdown />` adında bir component'e yapıştırıcağız. Oluşturacağımız bu component'in bu değerini kestiğimiz kısma koyabiliriz.

Devamında terminalimize gidiyoruz. Component'i terminalden oluşturacağız. Aşağıda yazmamız gereken komut bulunuyor.

```
php artisan make:component CategoryDropdown
```

Komutu çalıştırdıktan sonrasında **components** klasörümüzün altında component'imizin geldiğini görebiliyoruz. Belgenin içerisine girip bütün içeriği silip kestiğimiz değeri yapıştırıyoruz.

- Peki, biz terminalden `component` oluşturduğumuzda bize bir artısı bulunuyor mu?

Bu sorunun cevabı evet şu anda **app/View/Components/** yoluna giderseniz oluşturduğumuz component için bize bir sınıf oluşturduğunu görebiliyoruz. Bu sınıfta iki adet metot bulunuyor birisi sınıf başladığında otomatik olarak çalıştırılan metot diğer metot `render` adında bize view işlemini yapıyor. Şu anda sayfada `$categories` adında bir değişkenin tanımsız olduğunu gösteriyor. Daha demin view işlemi yapıtığını söylemiştim buraya değişkenimizi ekleyebiliriz. Aşağıda son hali bulunuyor.

```
return view('components.category-dropdown', [
    'categories' => Category::all()
]);
```

Bu işlemi yaptıktan sonrasında sayfamızın çalıştığını görebiliyoruz. Şu anda sınıfın üstüne bulunan kodlar bizim işimize yaramıyor bu yüzden silebiliriz.

Artık **routes/web.php** belgemizde bulunan authors route'ının `'categories' => Category::all()` kısmını silebiliriz. Ve aynı şekilde **PostController.php** belgemizde bulunan `index` metotumuzun içerisinde bulunan `view` işlevinin içerisinde ki `'categories' => Category::all()` değerinide silebiliriz.

8

Sayfalandırma

Bu Bölümde

Bu bölümde, Post'larımızın 500 adet olduğunu düşünelim. Bunların alt alta olması gerçekten kötü bir görüntü yaratacaktır. Bu yüzden **sayfa 1, sayfa 2, sayfa 3…** mantığında sayfalandırmamız en sağlıklısı olacaktır.

Laravel bize bu işlem için **pagination** özelliğini sunar.

Basit Sayfalandırma

İlk olarak 8. Bölümün girişinde söyledilerimizi. Orada anlatılanı anladığımıza göre başlayabiliriz. Dediğim gibi laravel bize **pagination** özelliğini sunuyor, özelliğin kendisinin içerisine yazdığınız sayı kadar bir sayfada o kadar Post bulunuyor.

`paginate(2)` kodumuz ile bir sayfada iki adet Post bulunur. Eğer ki toplamda otuz adet Post var ise on beş adet sayfa bulunacaktır.

O zaman `pagination` özelliğimizi kullanalım. Bunun için **PostController.php** belgemize gidelim. Belgemizin içerisinde `index` metotumuz bulunuyor, metotumuzun içerisinde bulunan `view()` işleminin en sonunda bulunan `get()` kısmını `paginate(3)` olarak değiştiriyoruz. Son hali aşağıda bulunuyor.

```
return view('posts.index',[
    'posts' => Post::latest()->filter(
        request(['search', 'category'])
    )->paginate(6)
]);
```

Şu anda sayfamız da gerçekten altı adet Post bulunuyor, yani çalışıyor. Bu güzel fakat diğer sayfaları göremiyoruz. Aslında `page` adında bir `get` parametresini kullanarak diğer sayfaları görebiliriz. Örnek olarak aşağıdaki linki kullanırsanız ikinci sayfayı göreceksiniz. `/?page=2`

Tamam bu bizim için güzel fakat kullanıcıyı link ile uğraştırmak istemeyiz. Eğerki **index.blade.php** belgemize gidersek burada bir `if else` işlemi yapmıştık. `İf` bloğunun içerisine `{{ $posts->links() }}` yazarsanız başarılı bir şekilde sayfanın aşağı kısımlarında sayfanın numarandırıldığını göreceksiniz.

Şu anda sayfada eksiklerimiz bulunuyor. Bunların en basit örneği şu, siz eğer ki herhangi bir sayfada olduğunuzda `category` kısmından herhangi bir `category` seçtiğinizde size sonuç göstermiyor.

Bu problemi çözmek için ilk olarak **category-dropdown.blade.php** belgemize gidelim. Belgede daha öncesinde `http_build_query` işlemi yapmıştık. İçerisinde bulunan `'category'` değerinin yanına `'page'` değerini ekleyelim.

Devamında `x-dropdown-item` component'imizin href kısmını:

```
/?{{http_build_query(request()->except('category', 'page')) }}
```

olarak değiştirelim. Şu anda başarılı bir şekilde çalışıyor.

9

Formlar ve Kimlik Doğrulama

Bu Bölümde

Bu bölümde, artık projemizin son işlemini yapmaya hazırız. Başlıkta da yazdığı gibi kimlik doğrulama ve formlar üzerinde çalışacağız.

Kullanici Kayit Sayfasi Oluşturma

İlk olarak kullanıcıların kayıt olması için bir link oluşturalım.

```
/register
```

Fakat böyle bir rotamız olmadığı için maalesef bir sonuç alamayacağız. Bunun için ilk olarak **routes/web.php** belgemize gidiyoruz. Oluşturduğumuz rotanın **RegiterController** adında bir controller'ın içerisinde bulunan `create` metotunu çalıştırmasını istiyoruz. Rota yapısının son yapısı aşağıda bulunuyor.

```
Route::get('register', [RegisterController::class, 'create']);
```

Şu anda tahmin edebileceğiniz gibi çalışmayacaktır. Sebebi böyle bir controller'ımız bulunmuyor. Bunun için terminalimizi açıyoruz. Devamında make'i kullanarak controller'ımızı oluşturuyoruz.

`php artisan make:controller RegisterController` Devamında oluşturduğumuz controller'ımızı `use \App\Http\Controllers\RegisterController;` bu şekilde sayfamıza dahil ediyoruz. Sırada `create` metotumuzu eklememiz gerekiyor. Bunun için RegisterController'ımıza gidiyoruz.

Belgenin içerisinde olması gerektiği gibi `RegisterController` adında bir sınıf bulunuyor. İçerisine şimdilik hello world değerini döndüren create metotumuzu oluşturuyoruz.

```
public function create(){
return 'hello world';
}
```

Şu anda sayfamızı yenilediğimizda başarılı bir şekilde `hello world` değerini alıyoruz.

Sırada sayfanın yapısını içerisine eklememiz gerekiyor. Metotumuzun içerisinde bulunana `hello world` yazısı yerine view işlemini yapacağız. Kendisi resources dosyamızın altında bulunan views dosyamızın içerisinde oluşturacağımız register klasörünün içindeki **create.blade.php** belgesinin içeriğini alacak.

```
return view('register.create')
```

Devamında bahsettiğim yola **register/create.blade.php** dosyamızı ve belgemizi oluşturuyoruz. Sonrasında **create.blade.php** belgemizin içerisine giriyoruz. Sayfamızın yapısının aslında **show.blade.php** belgemizin ilk iki satırını kopyalayarak alabiliriz.

```
<x-layout>
    <section class="px-6 py-8">
```

Bunları **create.blade.php** belgemizin içerisine alabiliriz. Tabikide kendilerini bu şekilde kullanamayız. Altlarını kapatmamız gerekiyor.

```
<x-layout>
    <section class="px-6 py-8">
    </section>
</x-layout>
```

Sayfada konumunu görmek için örnek olarak section'larımızın arasına `<h1>Hello World</h1>` yazalım. Şu anda sayfamıza baktığımız başarılı bir şekilde sonuç alabiliyoruz. O zaman form'umuzu burada oluşturabiliriz. İlk olarak class değeri:

```
max-w-lg mx-auto mt-10 bg-gray-100 p-6 border border-gray-200
 rounded-xl
```

olan bir main oluşturuyoruz.

```
<main class="max-w-lg mx-auto mt-10 bg-gray-100 p-6 border border-gray-200
rounded-xl
">
</main>
```

İçerisine POST işlemi yapan register adında bir action'u olan bir form oluşturalım.

```
<form method="POST" action="/register" class="mt-10"
>

</form>
```

Devamında form'un içerisinde bir adet div oluşturacağız. Div'imizin içerisinde bir adet **label** ve **input** bulunacak. İnput'umuzun tipi text ve doldurulması zorunlu olacak. Label'ımızın içerisinde de Username değeri bulunacak. Bütün elemanların CSS değerleri olacak. Fakat biz burada Laravel yazdığımız için CSS kısmını anlatmamız mantıklı olmayacaktır. Son olarak sayfamızın başlığı olması adında form'umuzun üstünde `h1` etiketimizin içerisinde `Register!` değerini oluşturuyoruz.

Oluşturduğumuz **create.blade.php** belgemizin içeriğinin son hali aşağıdaki görselde yer almaktadır.

PHP Dosyasının Yolu

`resources/views/register/create.blade.php`

```
<x-layout>
    <section class="px-6 py-8">
        <main class="max-w-lg mx-auto mt-10 bg-gray-100 border border-gray-200 p-6 rounded-xl">
            <h1 class="text-center font-bold text-xl">Register!</h1>

            <form method="POST" action="/register" class="mt-10">
                <div class="mb-6">
                    <label class="block mb-2 uppercase font-bold text.xs text-gray-700"
                        for="username"
                    >
                        Username
                    </label>

                    <input class="border border-gray-400 p-2 w-full"
                        type="text"
                        name="username"
                        id="username"
                        required
                    >
                </div>
            </form>
        </main>
    </section>
</x-layout>
```

Şu anda sayfamıza baktığımızda başarılı bir şekilde oluşturduğumuz form'u ve içindeki elemanı da görebiliyoruz. Biliyorsunuz ki veritabanımızda bulunan **users** tablomuzun içerisinde `username` kolonundan başka değerlerde bulunuyor. Yapmamız gereken şey onlarıda sayfamıza eklemek.

Aslında form'umuzun içerisinde bulunan div'in kendisi ve içeriği bizim bu işimizi görüyor. Sadece üstünde ufak değişiklikler yapmamız gerekiyor. O zaman bu div'imizi kopyalayalım ve kendisinin üstüne yapıştıralım. Yapıştırdığımız elemanın kendisi kullanıcının name değerini alacak. Bu yüzden div'in içerisindeki bazı değerleri değiştirmemiz gerekiyor.

Giriş Sayfasını Oluşturma

Artık bir kullanıcının kendi hesabına giriş yapması için bir form oluşturma zamanı geldi. Bunun için ilk olarak önceki bölümün sonlarında eklediğimiz Log In bölümüne tıklayalım.

Evet **404 Not Found** hatası aldık. O zaman ilk olarak login adında bir rota oluşturalım. Bunun için ilk olarak **routes/web.php** belgemize gidelim. Belgemizde bulunan logout rotamızın üstüne kendisinden bir tane daha kopyalayıp yapıştıralım.

Sonrasında logout olan bölümünü login olarak değiştirelim ve tahmin edebileceğiniz gibi `destroy` metotumuz da değiştirmemiz gerekiyor. Kendisinin isimini `create` olarak değiştirelim. Rotamızın son hali aşağıda bulunuyor.

```
Route::get('login', [SessionController::class, 'create']);
```

Devamında **login** ve **logout** rotalarımıza `middleware` ekleyelim. Her iki rotamızın da son hali aşağıda bulunuyor.

```
Route::get('login', [SessionController::class, 'create'])->middleware('guest');
Route::post('logout', [SessionController::class, 'destroy'])->middleware('auth');
```

Devamında SessionController'ımıza gidip `destroy` metotumuzun üstüne `create` adında metotumuzu oluşturalım. Oluşturduğumuz metot bizi session klasörünün altında create adında bir belgeye yönlendirsin. Metotumuzun son hali aşağıda bulunuyor.

```
public function create() {
    return view('sessions.create');
}
```

Devamında views klasörümüzün altına session klasörümüzü ve içerisine **create.blade.php** belgesini oluşturalım. Belgemizin içerisine sayfamızın temellerini yazalım.

```
<x-layout>
    <section class="px-6 py-8">

      <main class="max-w-lg mx-auto mt-10 bg-gray-100 border border-gray-200
p-6 rounded-xl">
```