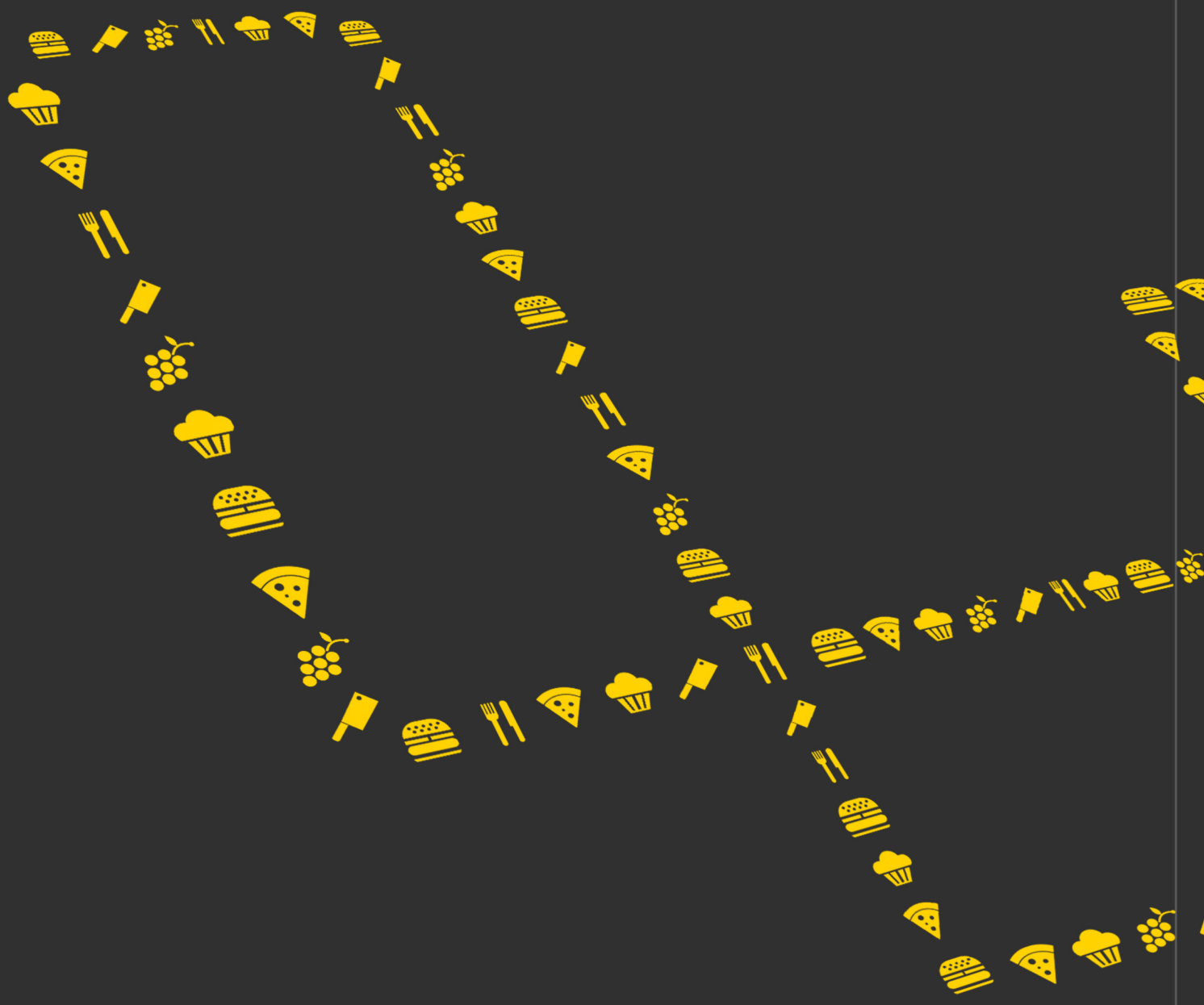

LARAVEL 4
COOKBOOK

YAZARLAR CHRISTOPHER PITT VE SİNAN ELDEM

Laravel 4 Cookbook (TR)

Laravel 4 öğrenmek için inşa edebileceğiniz projeler.

Christopher Pitt, Taylor Otwell ve Sinan Eldem

Bu kitap şu adreste satılmaktadır http://leanpub.com/laravel4cookbook-tr

Bu versiyon şu tarihte yayımlandı 2014-06-19

Kitabı tweetleyin!

Christopher Pitt, Taylor Otwell ve Sinan Eldem'a kitabını şu adresten Twitter tanıtarak yardımcı olun!

Kitap için önerilen tweet:

Laravel 4 Cookbook kitabının Türkçe Çevirisi http://leanpub.com/laravel4cookbook-tr #laravel4cookbook-tr #laravel @laraveltr

Kitap için önerilen hashtag #laravel4cookbook-tr.

Kitap için diğerleri ne demiş merak ediyorsanız bağlantıya tıklayarak hashtagları arayabilirsiniz:

https://twitter.com/search?q =#laravel4cookbook-tr

İçindekiler

Laravel 4'ün Yüklenmesi

Laravel 4, bağımlılıklarını yönetmek için Composer kullanır. **http://getcomposer.org/doc/00-intro.md#installatio**
nix adresindeki talimatları izleyerek Composer'i yükleyebilirsiniz.

Composer'i çalıştırdıktan sonra, yeni bir klasör oluşturun veya mevcut bir klasöre girin ve aşağıdaki komutu kullanarak Laravel 4'ü yükleyin:

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

Şayet Composer'i global tarzda yüklemeyi seçmemişseniz (öyle yapmanız gerekmesine rağmen), bu durumda kullanmanız gereken komut şuna benzer olmalıdır:

```
1 php composer.phar create-project laravel/laravel ./ --prefer-dist
```

Bu komutların her ikisi de Laravel 4 yüklenme sürecini başlatacaktır. Bulunması ve indirilmesi gereken birçok bağımlılık vardır; bu nedenle bu sürecin tamamlanması biraz zaman alabilir.

Authentication (Kimlik Doğrulama)

Siz de benim gibiyseniz, şifre korumalı sistemler inşa ederken büyük bir zaman harcamış olmalısınız. Ben bir CMS (İçerik Yönetim Sistemi) veya alışveriş kartı için kimlik doğrulama sistemi ekleme noktasında dehşete düşerdim. Laravel 4 ile bunun ne kadar kolay olduğunu öğrenene kadar bu böyleydi.

> Bu bölümle ilgili kod şurada bulunabilir: **https://github.com/sineld/tutorial-laravel-4-authentication**

> Bu eğitici ders PHP 5.4 veya daha üstünü ve PDO/SQLite uzantısını gerektirir. Ayrıca Laravel 4'ün karşıladığı tüm gereksinimlere de sahip olmanız gerekiyor. Bunların bir listesini **http://laravel.gen.tr/docs/installation#server-requirements** adresinde bulabilirsiniz.

Laravel'in Yüklenmesi

Laravel 4 bağımlılıklarını yönetmek için Composer kullanır. **http://getcomposer.org/doc/00-intro.md#installation**
nix adresindeki talimatları izleyerek Composer'i yükleyebilirsiniz.

Composer yüklemesinden sonra yeni bir dizin oluşturun veya mevcut bir dizine gidin ve aşağıdaki komutu kullanarak Laravel'i yükleyin:

```
 composer create-project laravel/laravel .

Installing laravel/laravel (v4.1.27)
...
```

Eğer Composer'i global olarak yüklemeyi seçmemişseniz (gerçekte öyle yapmanız gerektiği halde), bu durumda kullanmanız gereken komut şuna benzer olmalıdır:

```
 php composer.phar create-project laravel/laravel .

Installing laravel/laravel (v4.1.27)
...
```

Bu komutların her ikisi de Laravel 4 yüklenme sürecini başlatacaktır. Bulunması ve indirilmesi gereken birçok bağımlılık vardır; bu nedenle bu sürecin tamamlanması biraz zaman alabilir.

> Bu dersin temel aldığı Laravel sürümü **4.1.27** dir. Laravel'in sonraki sürümlerinde burada gösterilen koda değişiklikler olması muhtemeldir. Bu durumda, yukarıda söz edilen Github ambarını klonlayın ve `composer install` komutunu çalıştırın. Sizin için Laravel **4.1.27** yüklenecek şekilde bir lock dosyası dahil edilmiştir.

Veritabanının Yapılandırılması

Kullanıcıları ve kimlik doğrulamayı yönetmenin en iyi yollarından birisi bunların bir veritabanında saklanmasıdır. Ön tanımlı Laravel 4 authentication bileşenleri bir veritabanı deposu kullanacağınızı varsayar ve bu veritabanı kullanıcılarının getirileceği ve kimlik doğrulamasının yapılabileceği iki sürücü sağlamaktadır.

Veritabanına Bağlanma

Sağlanan sürücülerden herhangi birisini kullanmak için, öncelikle o veritabanına geçerli bir bağlantıya ihtiyacımız var. Bunu `app/config/database.php` dosyasındaki ilgili kesimlerin yapılandırılması yoluyla ayarlıyoruz. Benim test amaçlı kullandığım SQLite veritabanı için bir örnek şöyledir:

```php
<?php

return [
  "fetch"       => PDO::FETCH_CLASS,
  "default"     => "sqlite",
  "connections" => [
    "sqlite" => [
      "driver"   => "sqlite",
      "database" => __DIR__ . "/../database/production.sqlite"
    ]
  ],
  "migrations"  => "migration"
];
```

Bu dosya `app/config/database.php` olarak kaydedilmelidir.

Ben yorumları, ilgisiz satırları ve gereksiz sürücü yapılandırma seçeneklerini çıkarttım.

Bu dersin önceki sürümleri user deposu için bir MySQL veritabanı kullanmıştı. Laravel kodu gösterimi yaparken yüklemesi ve kullanması basit olduğu için bunu SQLite olarak değiştirmeye karar verdim. Eğer migrasyonlar kullanıyorsanız bu sizi hiç etkilemeyecektir. SQLite hakkında daha fazlasını **https://laracasts.com/lessons/maybe-you-should-use-sqlite** adresinden öğrenebilirsiniz.

Veritabanı Sürücüsü

Laravel 4'ün sağladığı ilk sürücü **database** adındadır. Adının düşündürdüğü gibi, bu sürücü verilen kimlik bilgilerine (credentials) uyan kullanıcıların mevcut olup olmadığını ve verilen kimlik doğrulama bilgilerinin uygun olup olmadığını tayin etmek için veritabanını doğrudan sorgular.

Eğer bu sürücüyü kullanmak istiyorsanız; veritabanınızda aşağıdaki veritabanı tablosunun yapılandırılmış olması gereklidir:

```
CREATE TABLE user (
  id integer PRIMARY KEY NOT null,
  username varchar NOT null,
  password varchar NOT null,
  email varchar NOT null,
  remember_token varchar NOT null,
  created_at datetime NOT null,
  updated_at datetime NOT null
);
```

Burada ve bundan sonrasında, çoğul veritabanı tablo isimlendirme standardından ayrılıyorum. Genellikle, standartlara yapışmanızı önereceğim fakat bu bana hem migrasyonlarda hem de mo-

> dellerde veritabanı tablo isimlerini nasıl yapılandırabileceğinizi göstermek için bir fırsat veriyor.

Eloquent Sürücüsü

Laravel 4'ün sağladığı ikinci sürücü **eloquent** adındadır. Eloquent aynı zamanda Laravel'in model verisinin soyutlanlaması için sağladığı ORM'nin de adıdır. Bir kullanıcının doğrulanmış olup olmadığını tayin etmek için nihayetinde bir veritabanını sorgulama noktasında benzerdir fakat tayin etmek için kullanacağı arayüz, doğrudan veritabanı sorgularından oldukça farklıdır.

Şayet Laravel 4 kullanarak orta-büyük ölçekli uygulamalar inşa ediyorsanız, veritabanı nesnelerini temsil etmek için Eloquent modelleri kullanmak gibi iyi bir şansınız var. Bunu aklımızda tutarak, authentication sürecinde Eloquent modellerinin yer alması üzerine bir zaman ayıracağım.

> Eğer Eloquent ile ilgili tüm şeyleri gözardı etmek istiyorsanız, modellerle ilgili aşağıdaki kesimleri atlayabilirsiniz.

Bir Migrasyon Oluşturulması

Uygulamamızın veritabanıyla iletişimini yönetmek için Eloquent kullanacağımız için, Laravel'in veritabanı tablo manipulasyon araçlarını pekala kullanabiliriz.

Başlamak için, projenizin kök dizinine gelin ve aşağıdaki komutu yazın:

```
 php artisan migrate:make --table="user" create_user_table

Created Migration: 2014_05_04_193719_create_user_table
Generating optimized class loader
Compiling common classes
```

> Buradaki `--table` etiketi, `User` modelinde tanımlayacağımız `$table` özelliğine tekabül etmektedir.

Bu komut user tablosu için aşağıdaki gibi bir kalıp üretecektir:

```php
<?php

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

class CreateUserTable
  extends Migration
{
  public function up()
  {
    Schema::table("user", function(Blueprint $table) {

    });
  }

  public function down()
  {
    Schema::table("user", function(Blueprint $table) {

    });
  }
}
```

> Bu dosya `app/database/migrations/****_**_**_******_create_user_table.php` şeklinde kaydedilecektir. Yıldızlar diğer sayılarla değişeceği için sizinkiler biraz farklı olabilir.

Bu dosya isimlendirme düzeni biraz garip gözükebilir ama iyi bir sebebi vardır. Migration sistemleri her türlü serverde çalışabilecek şekilde tasarlanır ve hangi sırayla çalışacakları sabit olmalıdır. Bütün bunlar veritabanı değişikliklerinin versiyon kontrollü olmasını sağlar.

Migration sadece çok temel bir kod kalıbı üretir, yani user tablosu için bizim alanlar eklememiz gerekir:

```php
<?php

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

class CreateUserTable
  extends Migration
{
  public function up()
  {
    Schema::create("user", function(Blueprint $table) {
      $table->increments("id");
      $table->string("username");
      $table->string("password");
      $table->string("email");
      $table->string("remember_token")->nullable();
      $table->timestamps();
    });
  }

  public function down()
  {
    Schema::dropIfExists("user");
  }
}
```

> Bu dosya `app/database/migrations/****_**_**_******_create_user_table.php` şeklinde kaydedilecektir.

Burada biz `id`, `username`, `password`, `created_at` ve `updated_at` için alanlar ekledik. Ayrıca, migrasyonlar geri alınacaksa çalıştırılacak olan bir drop metodu da ekledik; bu metod, eğer mevcutsa user tablosunu drop edecektir.

Bu migrasyon siz sadece database sürücüsü kullanmak isteseniz dahi çalışacaktır, ancak genellikle modeller ve seeder'ları da içeren daha büyük bir işin parçasıdır.

> Schema hakkında daha fazlasını şurada bulabilirsiniz: **http://laravel.gen.tr/docs/schema**.

Bir Model Oluşturulması

Laravel 4, gerekli tüm interface metodları olan bir `User` modeli sağlamaktadır. Ben onu biraz değiştirdim ama temelleri hala duruyor...

```php
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User
  extends Eloquent
  implements UserInterface, RemindableInterface
{
  protected $table = "user";
  protected $hidden = ["password"];

  public function getAuthIdentifier()
  {
    return $this->getKey();
  }

  public function getAuthPassword()
  {
    return $this->password;
  }

  public function getRememberToken()
  {
    return $this->remember_token;
  }

  public function setRememberToken($value)
  {
    $this->remember_token = $value;
  }

  public function getRememberTokenName()
  {
    return "remember_token";
  }

  public function getReminderEmail()
```

```
39    {
40      return $this->email;
41    }
42  }
```

Bu dosya `app/models/User.php` olarak kaydedilmelidir.

`$table` özelliğini bizim tanımlamış olduğumuzu unutmayın. Bu, migrasyonlarımızda tanımladığımız tablo ile aynı olmalıdır.

`User` modeli `Eloquent` genişletmesidir ve modelin kimlik doğrulama ve hatırlatıcı işlemleri bakımından geçerli olmasını temin eden iki interface'i implemente etmektedir. Biz bu interface'lere daha sonra bakacağız ama bunun önemi bu interface'lerin gerektirdiği metodlara dikkat çekmektir.

Laravel bir kullanıcıyı tanımlamak için email adresi veya username'den birinin kullanılmasına ancak `getAuthIdentifier()` ile farklı bir alan döndürülmesine izin verir. **UserInterface** interface'i password alan ismini belirtir, ama bu, `getAuthPassword()` metodu override edilerek/değiştirilerek değiştirilebilmektedir.

Reminder token metodları hesaba özgü güvenlik tokenlerini oluşturmak ve doğrulamak için kullanılır. Daha ince ayrıntıları başka bir derse bırakmak en iyisi olacak...

`getReminderEmail()` metodu kullanıcıyla bir şifre reset emaili aracılığıyla iletişim kurmak için gerekli olan bir email adresi döndürür.

Diğer açılardan her türlü model özelleştirme işlemleri yapmakta serbestsiniz, yerleşik kimlik doğrulama mekanizmalarını bozma korkusu taşımayın.

Bir Seeder Oluşturma

Laravel 4, ilk migrasyondan sonra veritabanınıza kayıtlar eklemek için kullanılabilecek bir seeding sistemi de bulundurmaktadır. Projeme ilk kullanıcıları eklemek için bende aşağıdaki seeder sınıfı var:

```php
<?php

class UserSeeder
  extends DatabaseSeeder
{
  public function run()
  {
    $users = [
      [
        "username" => "christopher.pitt",
        "password" => Hash::make("7h3iMOST!53cu23"),
        "email"    => "chris@example.com"
      ]
    ];

    foreach ($users as $user) {
      User::create($user);
    }
  }
}
```

Bu dosya `app/database/seeds/UserSeeder.php` olarak kaydedilmelidir.

Bunun çalıştırılması benim kullanıcı hesabımı veritabanına ekleyecektir ancak bunu çalıştırabilmek için onu ana **DatabaseSeeder** sınıfına eklememiz gerekiyor:

```php
<?php

class DatabaseSeeder
  extends Seeder
{
  public function run()
  {
    Eloquent::unguard();
    $this->call("UserSeeder");
  }
}
```

Bu dosya `app/database/seeds/DatabaseSeeder.php` olarak kaydedilmelidir.

`DatabaseSeeder` sınıfı çağrıldığı zaman user tablosuna benim hesabımı koyacaktır. Şayet migrasyon ve modelinizi daha önceden ayarlamış ve geçerli veritabanı bağlantı detaylarını sağlamışsanız, o zaman aşağıdaki komutlar her şeyi yapacak ve çalışacaktır:

```
 composer dump-autoload

Generating autoload files

 php artisan migrate

Migrated: ****_**_**_******_create_user_table

 php artisan db:seed

Seeded: UserSeeder
```

İlk komut oluşturduğumuz tüm yeni sınıfların sınıf otomatik yükleyicisi (class autoloader) tarafından seçilmesini garantiye alacaktır. İkincisi migrasyon için belirtilen veritabanı tablolarını oluşturur. Üçüncü ise user tablosuna user verisini yerleştirecektir.

Authentication Yapılandırılması

Authentication bileşenleri için yapılandırma seçenekleri azdır ama bazı özelleştirmeler yapmamıza imkan verecektir.

```php
<?php

return [
  "driver"   => "eloquent",
  "model"    => "User",
  "reminder" => [
    "email"  => "email/request",
    "table"  => "token",
    "expire" => 60
  ]
];
```

Bu dosya `app/config/auth.php` olarak kaydedilmelidir.

Bu ayarların hepsi önemlidir ve çoğu kendini açıklayıcıdır. İstek emailini oluşturmakta kullanılan görünüm `email` anahtarı ile belirtiliyor ve reset tokeninin sonlanma zamanı `expire` anahtarı ile belirtiliyor.

`email` anahtarı ile belirtilen view'e özel dikkat gösterin—Bu, Laravel'e ön tanımlı `app/views/emails/auth/reminder.bla` yerine `app/views/email/request.blade.php` dosyasını yüklemesini söylemektedir.

Giriş Yapma

Kimliği doğrulanmış kullanıcıların uygulamamızı kullanmasına izin vermek için, bir login sayfası inşa edeceğiz; orada kullanıcılar login detaylarını girebilecekler. Eğer detayları geçerli ise profil sayfalarına yönlendirilecekler.

Bir Layout View Oluşturulması

Uygulamamız için herhangi bir sayfa oluşturmadan önce düzen biçimlendirme ve stillerimizin tamamını soyutlamak akılcı olacaktır. Bu amaçla, Blade şablon motorunu kullanarak çeşitli include'leri olan bir layout view oluşturacağız.

Her şeyden önce layout view oluşturmamız gerekiyor:

```
1  <!DOCTYPE html>
2  <html lang ="en">
3    <head>
4      <meta charset ="UTF-8" />
5      <link rel ="stylesheet" href ="/css/layout.css" />
6      <title>Tutorial</title>
7    </head>
8    <body>
9      @include("header")
10     <div class ="content">
11       <div class ="container">
12         @yield("content")
13       </div>
```

```
    </div>
    @include("footer")
  </body>
</html>
```

Bu dosya `app/views/layout.blade.php` olarak kaydedilmelidir.

Bu layout view'in çoğu kısmı standard HTML olup içinde Blade'e özgü iki tag bulunmaktadır. Bunlardan `@include` tagları Laravel'e views klasöründen ilgili view'leri (`header` ve `footer` gibi stringler olarak isimlendirilmiş) dahil etmesini söyler.

`.blade.php` uzantısını göz ardı ettiğimize dikkat ediniz. Laravel bunu bizim için otomatik olarak ekler. Ayrıca, layout view'e her iki include'i dahil etmek için sağlanan veriyi de bağlar.

İkinci Blade tagı `@yield` dir. Bu tag bir kesim (section) ismi kabul eder ve o kesimde saklanan veriyi çıktılar. Bizim uygulamamızdaki view'ler bu layout view'i genişletecektir; kendi `content` kesimlerini belirteceğiz, böylece bunların markup kodu layout'un markup koduna gömülecektir. Kesimlerin tam olarak nasıl tanımlanacağını birazdan göreceksiniz.

```
@section("header")
  <div class ="header">
    <div class ="container">
      <h1>Tutorial</h1>
    </div>
  </div>
@show
```

Bu dosya `app/views/header.blade.php` olarak kaydedilmelidir.

Header include dosyası iki blade tagı içeriyor, ikisi birlikte Blade'e isimli kesimdeki markup'ı saklaması ve şablonda onu göstermesi talimatını verir.

```
@section("footer")
  <div class ="footer">
    <div class ="container">
      Powered by <a href ="http://laravel.com">Laravel</a>
    </div>
  </div>
@show
```

Bu dosya `app/views/footer.blade.php` olarak kaydedilmelidir.

Aynı şekilde, footer da isimli bir kesimdeki markup'ı sarmalar ve şablonda onu gösterir.

Bu include dosyalarındaki, kesimlerdeki markup'ı neden sarmalamamız gerektiğini merak ediyor olabilirsiniz. Onları hemen gösteriyoruz, hepsi bu. Böyle yapmak onların içeriğini değiştirmemize imkan verir. Bunu biraz sonra eylemde göreceğiz.

```
body {
  margin      : 0;
  padding     : 0 0 50px 0;
  font-family : "Helvetica", "Arial";
  font-size   : 14px;
  line-height : 18px;
  cursor      : default;
}

a {
  color : #ef7c61;
}

.container {
  width    : 960px;
  position : relative;
  margin   : 0 auto;
}

.header, .footer {
  background  : #000;
  line-height : 50px;
  height      : 50px;
  width       : 100%;
```

```
  color       : #fff;
}

.header h1, .header a {
  display : inline-block;
}

.header h1 {
  margin      : 0;
  font-weight : normal;
}

.footer {
  position : absolute;
  bottom   : 0;
}

.content {
  padding : 25px 0;
}

label, input {
  clear  : both;
  float  : left;
  margin : 5px 0;
}
```

Bu dosya `public/css/layout.css` olarak kaydedilmelidir.

Bazı temel stiller ekleyerek bitiriyoruz; bunu `head` elementinde link etmiştik. Bunlar ön tanımlı font ve düzeni değiştirirler. Uygulamanız bunlar olmadan da çalışacaktır ama o zaman biraz karmaşık gözükecektir.

Bir Login View Oluşturulması

Login view aslında bir formdur ve kullanıcılar ona kimlik bilgilerini girerler.

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    {{ Form::label("username", "Username") }}
    {{ Form::text("username") }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
```

Bu dosya `app/views/user/login.blade.php` olarak kaydedilmelidir.

`@extends` tagı Laravel'e bu view'in layout view'i genişlettiğini anlatır. Sonraki `@section` tagı 'content' kesimine dahil edilecek markupı söyler. Bu taglar, oluşturacağımız (layout dışında) tüm viewlerin temelini oluşturmaktadır.

Daha sonra Laravel'e içindeki kodun `echo` cümlesi kullandığımızdaki gibi göstertilmesini istediğimizi anlatmak için `{{` ve `}}` kullanıyoruz. `Form::open()` metoduyla bir form açıyoruz; formun gönderileceği bir rota ve ikinci parametrede opsiyonel parametreler sağlıyoruz.

Daha sonra iki label ve üç input tanımlıyoruz. Labeller bir metin parametresiyle izlenen bir name parametresi alır. Text inputları bir name parametresi, bir varsayılan değer parametresi ve opsiyonel parametreler alır. Password inputları bir name parametresi ve opsiyonel parametreler kabul eder. Son olarak, submit inputu bir name parametresi ve bir metin parametresi (labeller gibi) kabul eder.

Ve, **Form::close()** çağırarak formu kapatıyoruz.

Laravel'in sunduğu **Form** metodları hakkında daha fazlasını şurada bulabilirsiniz: **http://laravel.gen.tr/docs/html**

Login view'imiz şimdi tamamdır ama inputu kabul eden ve bir sonuç döndüren sunucu taraflı kod olmadan işe yaramaz. Şimdi bunu halledelim!

Bu dersin önceki sürümlerinde bazı ekstra input nitelikleri ve onları desteklemeye yardım edecek bir JavaScript kitaplığına bir referans dahil edilmişti. Dersi basitleştirmek için bu nitelikleri çıkardım ama o scripti **http://polyfill.io** adresinde bulabilirsiniz.

Bir Login Eylemi Oluşturulması

Login eylemi, authentication mantığını, oluşturduğumuz görünümlere tutturan şeydir. Şayet takip ediyorsanız, bu şeylerden birini ne zaman bir browserde deneyeceğimizi merak ediyorsunuzdur. Bu noktaya kadar, uygulamamıza görünümü yüklemesini söyleyen bir şey yoktur.

Bunu yapmak için, login eylemi için bir rota eklememiz gerekiyor:

```
<?php

Route::any("/", [
  "as"   => "user/login",
  "uses" => "UserController@login"
]);
```

Bu dosya `app/routes.php` olarak kaydedilmelidir.

Routes dosyası bir view'i doğrudan göstertmek suretiyle, yeni bir Laravel 4 uygulaması için bir açış sayfası gösterir. Biz sadece bunu bir controller ve eylem kullanacak şekilde değiştirdik.

`as` anahtarı ile rota için bir isim belirtiyoruz ve `uses` anahtarı ile ona bir hedef konum veriyoruz. Bu, tüm çağrıları ön tanımlı `/` rotasına eşleştirecek ve bu rotaya kolaylıkla tekrar başvurmak için kullanabileceğimiz bir isme de sahiptir.

Daha sonra bir controller oluşturmamız gerekiyor:

```
<?php

class UserController
  extends Controller
{
  public function login()
  {
    return View::make("user/login");
  }
}
```

Bu dosya `app/controllers/UserController.php` olarak kaydedilmelidir.

`Controller` sınıfını genişleten bir `UserController` tanımlıyoruz. Bunun içinde, routes dosyasında belirttiğimiz `login()` metodunu da oluşturuyoruz. Bütün bunların şimdilik yaptığı tek şey login view'i browserda göstermektir ama ilerlememizi görebilmek için yeterlidir!

Kişisel kurulmuş bir web sunucunuz olmadığı sürece, büyük ihtimalle Laravel'in sağladığı yerleşik web sunucusunu kullanmak isteyeceksiniz. Teknik olarak bu sadece PHP 5.3 ile gelen kişisel web sunucusunun tepesinde frameworkün yüklenmesidir, ancak onu çalıştırmak için yine de aşağıdaki komutu vermeniz gerekiyor:

```
 php artisan serve

Laravel development server started on http://localhost:8000
```

Tarayıcınızda **http://localhost:8000** adresine gittiğiniz zaman, login sayfasını göreceksiniz. Eğer orada yoksa, büyük ihtimalle bu noktaya kadar bir şeyleri atlamışsınızdır.

Kullanıcıların Kimliklerinin Doğrulanması

Tamam, formumuzu aldık ve şimdi de kullanıcıları doğru bir biçimde doğrulayabilmek için onu veritabanına bağlamamız gerekiyor.

```
<?php

class UserController
  extends Controller
{
  public function login()
  {
    if ($this->isPostRequest()) {
      $validator = $this->getLoginValidator();

      if ($validator->passes()) {
        echo "Geçerlilik denetimini geçti!";
      } else {
        echo "Geçerlilik denetimini geçemedi!";
      }
    }

    return View::make("user/login");
  }

  protected function isPostRequest()
```

```
22    {
23      return Input::server("REQUEST_METHOD") == "POST";
24    }
25
26    protected function getLoginValidator()
27    {
28      return Validator::make(Input::all(), [
29        "username" => "required",
30        "password" => "required"
31      ]);
32    }
33  }
```

Bu dosya `app/controllers/UserController.php` olarak kaydedilmelidir.

`UserController` sınıfımız bir miktar değişti. İlk olarak `login()` metodunda post edilen veriler üzerinde işler yapmamız ve server `REQUEST_METHOD` özelliğini kontrol etmemiz gerekiyor. Eğer bu değer `POST` ise formun bu eyleme post edilmiş olduğunu kabul edebiliriz ve geçerlilik denetimi sürecine geçebiliriz.

> Aynı sayfa için `GET` ve `POST` eylemlerinin ayrı tutulması da sık yapılan bir şeydir. Böyle yapmak bir şeyleri daha düzenli bir hale getirir ve `REQUEST_METHOD` özelliğinin kontrol edilmesi gereğini ortadan kaldırır; ben her ikisini aynı eylemde halletmeyi tercih ediyorum.

Laravel `4` harika bir validation sistemi ve onun kullanımını kolaylaştıran `Validator::make()` metodunu sağlar. İlk argüman geçerlilik denetimi yapılacak bir veri dizisidir ve ikinci argüman bir kurallar dizisidir.

Biz sadece username ve password alanlarının gerekli olduklarını belirttik ama diğer birçok geçerlilik kuralı bulunmaktadır (bir kısmını kullanacağız). Bu `Validator` sınıfının ayrıca bir `passes()` metodu olup, gönderilen form verisinin geçerli olup olmadığını anlamak için kullanılır.

> Geçerlilik mantığını controllerin dışında saklamak kimi zaman daha iyidir. Ben çoğu keresinde onu bir modele koyarım ama siz özel olarak girdi işleyen bir validation (geçerlilik denetiminden geçiren) sınıfı da oluşturabilirsiniz.

Eğer formu post ederseniz, şimdi gerekli alanları sağlayıp sağlamadığımızı söyleyecektir, fakat bu tür mesajları göstermenin daha güzel bir yolu vardır...

```
public function login()
{
  $data = [];

  if ($this->isPostRequest()) {
    $validator = $this->getLoginValidator();

    if ($validator->passes()) {
      echo "Geçerlilik denetimini geçti!";
    } else {
      $data["error"] = "Username ve/veya password geçersizdir.";
    }
  }

  return View::make("user/login", $data);
}
```

Bu, `app/controllers/UserController.php` dosyasından alınmıştır.

Username veya password alanlarından herbiri için ayrı ayrı hata mesajları göstermek yerine biz her ikisi için tek bir hata mesajı gösteriyoruz. Login formları bu yolla biraz daha güvenlidir!

Bu hata mesajını gösterebilmek için login view'i de değiştirmemiz gerekiyor.

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    @if (isset($error))
      {{ $error }}<br />
    @endif
    {{ Form::label("username", "Username") }}
    {{ Form::text("username") }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
```

Bu dosya `app/views/user/login.blade.php` olarak kaydedilmelidir.

Büyük ihtimalle görmüş olabileceğiniz gibi, hata mesajının varlığı konusunda bir kontrol ekledik ve onu gösterdik. Eğer geçerlilik başarısız olursa, username alanının yukarısında hata mesajını görebileceksiniz.

Input İle Redirect İşlemi

Formların ortak sorunlarından birisi en sık yapılan form yeniden gönderileri için sayfanın nasıl yenileneceğidir. Bunun üstesinden bazı Laravel büyüleriyle geleceğiz. Post edilen form verilerini oturumda saklayacağız ve login sayfasına redirect yapacağız!

```
1  public function login()
2  {
3    if ($this->isPostRequest()) {
4      $validator = $this->getLoginValidator();
5
6      if ($validator->passes()) {
7        echo "Geçerlilik denetimini geçti!";
8      } else {
9        return Redirect::back()
10         ->withInput()
11         ->withErrors($validator);
12     }
13   }
14
15   return View::make("user/login");
16 }
```

Bu dosya `app/controllers/UserController.php` olarak kaydedilmelidir.

Hata mesajlarını view'e atamak yerine, post edilen input verisini ve geçerlilik hatalarını geçmek suretiyle aynı sayfaya geri yönlendiriyoruz. Bu aynı zamanda view'imizi de değiştirmemiz gerekeceği anlamına geliyor:

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("username", "Username") }}
    {{ Form::text("username", Input::old("username")) }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
```

Artık bizden formu yeniden gönderme izni istemesi olmaksızın, yenile düğmesine basabiliriz.

Verilen Kimlik Bilgilerinin Doğrulanması

Kimlik doğrulamada son adım, sağlanan form verilerinin veritabanından kontrol edilmesidir. Laravel bunu bizim için kolaylıkla halleder:

```
<?php

class UserController
  extends Controller
{
  public function login()
  {
    if ($this->isPostRequest()) {
      $validator = $this->getLoginValidator();

      if ($validator->passes()) {
        $credentials = $this->getLoginCredentials();

        if (Auth::attempt($credentials)) {
          return Redirect::route("user/profile");
        }

        return Redirect::back()->withErrors([
          "password" => ["Kimlik Bilgileri geçersizdir."]
        ]);
      } else {
        return Redirect::back()
          ->withInput()
```

```
24            ->withErrors($validator);
25        }
26      }
27
28      return View::make("user/login");
29    }
30
31    protected function isPostRequest()
32    {
33      return Input::server("REQUEST_METHOD") == "POST";
34    }
35
36    protected function getLoginValidator()
37    {
38      return Validator::make(Input::all(), [
39        "username" => "required",
40        "password" => "required"
41      ]);
42    }
43
44    protected function getLoginCredentials()
45    {
46      return [
47        "username" => Input::get("username"),
48        "password" => Input::get("password")
49      ];
50    }
51  }
```

Bu dosya `app/controllers/UserController.php` olarak kaydedilmelidir.

Yapmamız gereken tek şey post edilen form verilerini (`$credentials`) `Auth::attempt()` metoduna geçmektir ve eğer kullanıcı kimlik bilgileri (credentials) geçerli ise, o kullanıcı login yapmış olacaktır. Eğer geçerliyse, user profile sayfasına bir redirect döndürüyoruz.

Şimdi bu sayfayı inşa edelim:

```
@extends("layout")
@section("content")
  <h2>Merhaba {{ Auth::user()->username }}</h2>
  <p>Profil sayfanıza hoş geldiniz.</p>
@stop
```

Bu dosya `app/views/user/profile.blade.php` olarak kaydedilmelidir.

```
Route::any("/profile", [
  "as"   => "user/profile",
  "uses" => "UserController@profile"
]);
```

Bu, `app/routes.php` dosyasından alınmıştır.

```
public function profile()
{
  return View::make("user/profile");
}
```

Bu, `app/controllers/UserController.php` dosyasından alınmıştır.

Kullanıcı giriş yaptıktan sonra, `Auth::user()` metodunu çağırmak suretiyle onun kaydına erişebiliriz. Bu bir User model olgusu (eğer Eloquent auth sürücüsü kullanıyorsak) ya da eski düz PHP nesnesi (Database sürücüsü kullanıyorsak) döndürür.

Auth sınıfı hakkında daha fazlasını şurada bulabilirsiniz: **http://laravel.gen.tr/docs/security#authenticating-users**.

Şifrelerin Sıfırlanması

Laravel'de yerleşik bulunan şifre sıfırlama bileşenleri harikadır! Biz onu, kullanıcılar sadece email adreslerini sağlayarak şifrelerini sıfırlayabilecekleri bir şekilde ayarlayacağız.

Bir Şifre Sıfırlama View'i Oluşturulması

Kullanıcıların şifrelerini sıfırlayabilmeleri için iki view'e ihtiyacımız var. Bir sıfırlama tokeni gönderilebilmesi için kullanıcıların email adreslerini girebilecekleri bir view gerekiyor ve hesapları için yeni bir şifre girebilecekleri bir view gerekiyor.

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::old("email")) }}
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@stop
```

Bu dosya `app/views/user/request.blade.php` olarak kaydedilmelidir.

Bu view sadece bir email adresi alanına sahip olmak dışında login view'e benzerdir.

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    {{ $errors->first("token") }}<br />
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::get("email")) }}
    {{ $errors->first("email") }}<br />
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("password_confirmation", "Confirm") }}
    {{ Form::password("password_confirmation") }}
    {{ $errors->first("password_confirmation") }}<br />
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@stop
```

Bu dosya `app/views/user/reset.blade.php` olarak kaydedilmelidir.

Tamam, bu kadar. Bazı inputlar ve hata mesajları olan bir form var. Ayrıca, şifre token istek emaili view'ini hafifçe modifiye ettim, yine de yeni Laravel 4 yüklemeleriyle sağlanan ön tanımlı view ile çoğu yeri aynıdır.

```
<!DOCTYPE html>
<html lang ="en">
  <head>
    <meta charset ="utf-8" />
  </head>
  <body>
  <h1>Şifre Sıfırlama</h1>
    Şifrenizi sıfırlamak için bu formu tamamlayın:
    {{ URL::route("user/reset", compact("token")) }}
  </body>
</html>
```

Bu dosya `app/views/email/request.blade.php` olarak kaydedilmelidir.

Email için yapılandırma seçeneklerini ön tanımlı `app/views/emails/auth/reminder.blade.php` yerine bu view'i kullanacak şekilde değiştirdiğimizi hatırlayınız.

Bir Şifre Sıfırlama Eylemi Oluşturulması

Eylemlerin ulaşılabilir olabilmesi için rotalar eklememiz gerekiyor.

```
Route::any("/request", [
  "as"   => "user/request",
  "uses" => "UserController@request"
]);

Route::any("/reset/{token}", [
  "as"   => "user/reset",
  "uses" => "UserController@reset"
]);
```

Bu, app/routes.php dosyasından alınmıştır.

Unutmayın; request rotası bir sıfırlama tokeni istenmesi içindir ve reset rotası bir şifrenin sıfırlanması içindir. Ayrıca, artisan kullanarak bir şifre reset tokenleri tablosu oluşturmamız gerekiyor.

```
 php artisan auth:reminders-table

Migration created successfully!
Generating optimized class loader
```

Bu komut hatırlatıcı tablosu için bir migrasyon şablonu üretecektir:

```
<?php

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

class CreateTokenTable
  extends Migration
{
  public function up()
  {
    Schema::create("token", function (Blueprint $table) {
      $table->string("email")->index();
      $table->string("token")->index();
      $table->timestamp("created_at");
    });
  }

```

```
18    public function down()
19    {
20      Schema::drop("token");
21    }
22  }
```

Bu dosya `app/database/migrations/****_**_**_******_create_token_table.php` olarak kaydedilmelidir.

Laravel migrasyonları `app/database/migrations/****_**_**_******_create_password_reminders_table.php` olarak oluşturur ama ben `user` tablosu ile daha uygun olsun istedim. Şifre hatırlatıcı tablonuz `app/config/auth.php` dosyanızdaki `reminder.table` anahtarına uyduğu sürece bir sorun olmayacaktır.

Bundan sonra şifre sıfırlama eylemlerini eklemeye başlayabiliriz:

```
1   public function request()
2   {
3     if ($this->isPostRequest()) {
4       $response = $this->getPasswordRemindResponse();
5
6       if ($this->isInvalidUser($response)) {
7         return Redirect::back()
8           ->withInput()
9           ->with("error", Lang::get($response));
10      }
11
12      return Redirect::back()
13          ->with("status", Lang::get($response));
14    }
15
16    return View::make("user/request");
17  }
18
19  protected function getPasswordRemindResponse()
20  {
21    return Password::remind(Input::only("email"));
```

```
22  }
23
24  protected function isInvalidUser($response)
25  {
26    return $response === Password::INVALID_USER;
27  }
```

Bu kod `app/controllers/UserController.php` dosyasından alınmıştır.

Bu metodlar setindeki esas sihir `Password::remind()` metoduna yapılan çağrıdır. Bu metod, uyan bir kullanıcı var mı diye veritabanını kontrol eder. Eğer uyan birini bulursa o kullanıcıya bir email gönderilir, aksi takdirde bir hata mesajı döndürülür.

Bu hata mesajını karşılayacak bir reset view'i ayarlamamız gerekiyor:

```
1  @extends("layout")
2  @section("content")
3    {{ Form::open() }}
4      @if (Session::get("error"))
5        {{ Session::get("error") }}<br />
6      @endif
7      @if (Session::get("status"))
8        {{ Session::get("status") }}<br />
9      @endif
10     {{ Form::label("email", "Email") }}
11     {{ Form::text("email", Input::old("email")) }}
12     {{ Form::submit("reset") }}
13   {{ Form::close() }}
14 @stop
```

Bu dosya `app/views/user/request.blade.php` olarak kaydedilmelidir.

Bu rotayı ziyaret ettiğinizde, size bir email adresi ve bir submit düğmesi içeren bir form gösterilmelidir. Onun geçersiz bir email adresi ile tamamlanması bir hata mesajı göstertecektir, geçerli bir email adresi ile tamamlanması ise bir başarı mesajı göstertecektir. Yani email gönderilir...

Laravel, email gönderilmesi için bir ton yapılandırma seçeneği içerir. Şimdi onlarla devam etmek isterdim ama bu dersi odaklı tutmak adına size sadece `app/config/mail.php` dosyasındaki `pretend` anahtarını true olarak ayarlamanızı önereceğim. Bu, gerçekte o adımı atlamasına rağmen sanki uygulama email gönderiyor gibi davranacaktır.

```
public function reset($token)
{
  if ($this->isPostRequest()) {
    $credentials = Input::only(
      "email",
      "password",
      "password_confirmation"
    ) + compact("token");

    $response = $this->resetPassword($credentials);

    if ($response === Password::PASSWORD_RESET) {
      return Redirect::route("user/profile");
    }

    return Redirect::back()
      ->withInput()
      ->with("error", Lang::get($response));
  }

  return View::make("user/reset", compact("token"));
}

protected function resetPassword($credentials)
{
  return Password::reset($credentials, function($user, $pass) {
    $user->password = Hash::make($pass);
    $user->save();
  });
}
```

Bu kod `app/controllers/UserController.php` dosyasından alınmıştır.

`Password::remind()` metoduna benzer şekilde, `Password::reset()` metodu da kullanıcıya özgü verilerden oluşan bir dizi kabul eder ve bir demet sihir yapar. Bu sihir geçerli bir kullanıcı hesabı açısından kontrol edilmesi ve eşlik eden şifrenin değiştirilmesi ya da bir hata mesajı döndürülmesini içerir.

Bunun için reset view oluşturmamız gerekiyor:

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    @if (Session::get("error"))
      {{ Session::get("error") }}<br />
    @endif
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::old("email")) }}
    {{ $errors->first("email") }}<br />
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("password_confirmation", "Confirm") }}
    {{ Form::password("password_confirmation") }}
    {{ $errors->first("password_confirmation") }}<br />
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@stop
```

Bu dosya `app/views/user/reset.blade.php` olarak kaydedilmelidir.

Tokenler, `app/config/auth.php` dosyasında tanımlandığı gibi 60 dakika sonra sona erer.

Filtreler Oluşturulması

Laravel, tek bir rota veya rotalar grubu için çalışacak filtreler tanımlayabileceğimiz bir filters dosyası içermektedir. Bakacağımız en temel filtrelerden biri auth filtresidir:

```
<?php

Route::filter("auth", function() {
  if (Auth::guest()) {
    return Redirect::route("user/login");
  }
});
```

Bu dosya `app/filters.php` olarak kaydedilmelidir.

Bu filtre, rotaya uygulandığı zaman, kullanıcının hali hazırda giriş yapmış bir kullanıcı olup olmadığını görmek için kontrol yapacaktır. Eğer giriş yapmış değillerse, login rotasına yönlendirileceklerdir.

Bu filtreyi uygulamak için routes dosyamızda değişiklik yapmamız gerekiyor:

```
<?php

Route::any("/", [
  "as"   => "user/login",
  "uses" => "UserController@login"
]);

Route::group(["before" => "auth"], function() {

  Route::any("/profile", [
    "as"   => "user/profile",
    "uses" => "UserController@profile"
  ]);

});

Route::any("/request", [
  "as"   => "user/request",
  "uses" => "UserController@request"
```

```
]);

Route::any("/reset/{token}", [
  "as"   => "user/reset",
  "uses" => "UserController@reset"
]);
```

Bu dosya `app/routes.php` olarak kaydedilmelidir.

Profil rotasını bir auth filtesi çalıştıracak bir callback içine aldığımızı fark edeceksiniz. Bunun anlamı, profil rotası sadece kimliği doğrulanmış kullanıcılar için erişilebilir olacak demektir.

Bir Logout Eyleminin Oluşturulması

Bu yeni güvenlik önlemlerini test etmek için ve dersi tamamlamak için bir `logout()` metodu oluşturmamız ve kullanıcıların çıkış (log out) yapabilmeleri için header'e linkler eklememiz gerekiyor.

```
public function logout()
{
  Auth::logout();

  return Redirect::route("user/login");
}
```

Bu kod `app/controllers/UserController.php` dosyasından alınmıştır.

Bir kullanıcıya çıkış yaptırılması `Auth::logout()` metodunu çağırmak kadar kolay bir şeydir. Bu metod düşündüğünüz gibi session'ı temizlemez ama bizim `auth` filtremizin devreye girmesini sağlayacaktır.

Header dosyamızın yeni hali şöyle olacaktır:

```
@section("header")
  <div class ="header">
    <div class ="container">
      <h1>Tutorial</h1>
      @if (Auth::check())
        <a href ="{{ URL::route("user/logout") }}">
          logout
        </a> |
        <a href ="{{ URL::route("user/profile") }}">
          profile
        </a>
      @else
        <a href ="{{ URL::route("user/login") }}">
          login
        </a>
      @endif
    </div>
  </div>
@show
```

Bu dosya `app/views/header.blade.php` olarak kaydedilmelidir.

Son olarak, logout eylemi için bir rota eklemeliyiz:

```
Route::any("/logout", [
  "as"   => "user/logout",
  "uses" => "UserController@logout"
]);
```

Bu, `app/routes.php` dosyasından alınmıştır.