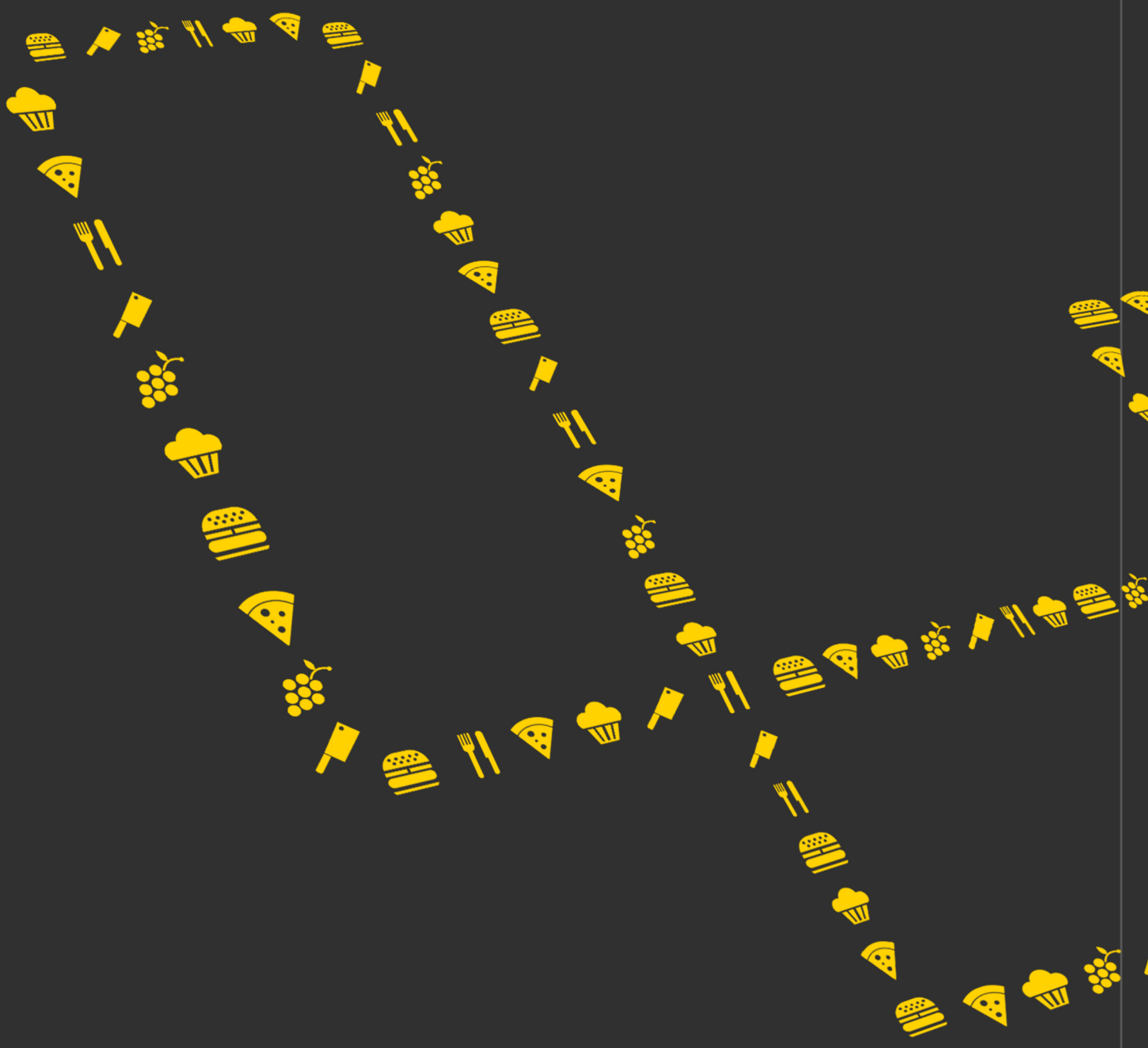

LARAVEL 4

LARAVEL4の美味しい料理法

BY CHRISTOPHER PITT 著　HIROHISA KAWASE 翻訳

Laravel 4 Cookbook 日本語版

Laravel4 の美味しい料理法

Christopher Pitt, Taylor Otwell and Hirohisa Kawase

This book is for sale at http://leanpub.com/laravel4cookbook-jp

This version was published on 2014-07-04

Tweet This Book!

Please help Christopher Pitt, Taylor Otwell and Hirohisa Kawase by spreading the word about this book on Twitter!

The suggested hashtag for this book is #laravel4cookbook-jp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

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

Contents

Laravel4 のインストール

Laravel4 は依存パッケージの解決に、Composer を使用しています。**http://getcomposer.org/doc/00-intro.md#installation-nix**の手順に従えば、Composer をインストールできます。

一度 Composer が動作すれば、新しいディレクトリーを作成するか、もしくは既存のディレクトリーを指定し、以下のコマンドで Laravel4 をインストールできます。[1]

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

もし、Composer をグローバルにインストールしていない場合、以下のコマンドが利用できます。（グローバルにインストールすることを推奨します。）

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

どちらのコマンドでも、Laravel4 のインストールが開始されます。多くの依存元があり、それらをダウンロードするため、終了するまで多少時間がかかるでしょう。

---

[1] Laravel ではコメントも解説ドキュメントの一部だと考えられています。もし、英語が苦手ならば、コメントを日本語に訳したインストール・設定パッケージも利用できます。その場合は”laravel/laravel” の代わりに、”laravel-ja/laravel” を指定してください。

認証

あなたも私と同じ種類の人間であれば、パスワード保護されたシステムを構築するために、莫大な時間を費やしてきたことでしょう。CMS やショッピングカートで、認証システムを使い、しっかりとネジを締めあげなくてはならない箇所を開発することをいつも恐れていました。Laravel4 で簡単に実現できることを学ぶまでは、ずっとそうでした。

> この章のコードは、**https://github.com/formativ/tutorial-laravel-4-authentication**で取得できます。

データベースを設定する

ユーザーと認証を管理する一つの良い方法は、データベースに項目を保存することです。デフォルトで Laravel4 の認証メカニズムは、データベースを使用することを前提としています。データベース上のユーザーを取得、認証するために、2つのドライバーが用意されています。

データベースに接続する

提供されている、両ドライバーを使うためには、まず有効なデータベース接続が必要です。**app/config/database.php** ファイルのセクションで、設定しましょう。

```
<?php

return [
    "fetch"       => PDO::FETCH_CLASS,
    "default"     => "mysql",
    "connections" => [
        "mysql" => [
                "driver"    => "mysql",
                "host"      => "localhost",
                "database"  => "tutorial",
                "username"  => "dev",
                "password"  => "dev",
                "charset"   => "utf8",
                "collation" => "utf8_unicode_ci",
                "prefix"    => ""
        ]
    ],
```

```
18      "migrations" => "migration"
19  ];
```

これは **app/config/database.php** ファイルとして保存してください。

コメント、余計な行、必要ないドライバー設定は削除しました。

データベースドライバー

Laravel4 が提供している最初のドライバーの名前は、**database** です。名前のとおり、指定された情報に一致するユーザーが存在するかどうかを決めるために、データベースへ直接クエリーします。

このドライバーを使用する場合、次のテーブルをデータベースに用意しておく必要があります。[2]

```
1  CREATE TABLE `user` (
2      `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
3      `username` varchar(255) DEFAULT NULL,
4      `password` varchar(255) DEFAULT NULL,
5      `email` varchar(255) DEFAULT NULL,
6      `created_at` datetime DEFAULT NULL,
7      `updated_at` datetime DEFAULT NULL,
8      PRIMARY KEY (`id`)
9  ) CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

本書の中では、テーブル名を複数型にする、Laravel4 のデフォルト命名ルールから逸脱しています。通常、私は標準的な方法にこだわるのですが、標準から外れることで、マイグレーションとモデルにおいて、データベース名をどのように設定するのかを明確にデモンストレーションしていきます。

[2] 実際にはメールの最大文字は 254 文字、パスワードはハッシュされ 64 文字必要です。varchar で定義するため、最大値の 255 で指定しているものと思われます。

Eloquent ドライバー

Laravel が提供している2つ目のドライバーは、**eloquent** と呼ばれています。Eloquent はデーターをモデルで抽象化するために、Laravel4 が提供している ORM の名前でもあります。

もしもあなたが Laravel4 で、中規模から大規模のアプリケーションを構築しているなら、データベースオブジェクトを表現するために Eloquent モデルを使用すれば、成功するチャンスが広がるでしょう。私は認証プロセスの Eloquent モデルに関連したコードを練り上げるために、時間を費やすことがあります。

> Eloquent に関することを全て無視したいのでしたら、以降のマイグレーションとモデルに関するセクションを飛ばしてお読みください。[a]

[a]マイグレーションは、Eloquent を使用しない場合でも、実際には使用できます。後ほど一言触れられています。

マイグレーションを作成する

データベースとアプリケーションがコミュニケーションをどのように取るかを管理するため、Eloquent を使用するのと同時に、Laravel4 のデータベース操作ツールも使用しましょう。

最初に、プロジェクトのルートへ移動し、次のコマンドを実行してください。

```
1  php artisan migrate:make --table="user" CreateUserTable
```

> **–table="user"** オプションは、**User** モデルの中で定義することになる、**$table="user"** プロパティに当たります。

この実行により、ユーザーテーブルの骨格となるコードが生成されます。以下のようなコードです。

```
1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4  use Illuminate\Database\Migrations\Migration;
5
6  class CreateUserTable
7  extends Migration
8  {
9      public function up()
10     {
11         Schema::table('user', function(Blueprint $table)
```

```
12          {
13              //
14          });
15      }
16      public function down()
17      {
18          Schema::table('user', function(Blueprint $table)
19          {
20              //
21          });
22      }
23  }
```

これは、**app/database/migrations/0000*0*000_000000_CreateUserTable.php** ファイルとして保存されます。ほとんどの 0 の部分は、他の数字に置き換わっているでしょう。

このファイルの名前は奇妙に思えるでしょうが、理由があります。マイグレーションシステムはどのサーバーでも動作するように設計されており、実行順序を固定するするために日時が先頭に付けられています。これにより、データーベース変更のバージョンコントロールを可能にしています。

マイグレーションは一番基本的な骨格だけを生成します。そのため、ユーザーテーブルにフィールドを追加記述する必要があります。

```
1   <?php
2
3   use Illuminate\Database\Schema\Blueprint;
4   use Illuminate\Database\Migrations\Migration;
5
6   class CreateUserTable
7   extends Migration
8   {
9       public function up()
10      {
11          Schema::create("user", function(Blueprint $table)
12          {
13              $table->increments("id");
14
15              $table
16                  ->string("username")
17                  ->nullable()
18                  ->default(null);
19
```

```
20              $table
21                  ->string("password")
22                  ->nullable()
23                  ->default(null);
24
25              $table
26                  ->string("email")
27                  ->nullable()
28                  ->default(null);
29
30              $table
31                  ->dateTime("created_at")
32                  ->nullable()
33                  ->default(null);
34
35              $table
36                  ->dateTime("updated_at")
37                  ->nullable()
38                  ->default(null);
39          });
40      }
41
42      public function down()
43      {
44          Schema::dropIfExists("user");
45      }
46  }
```

これは、**app/database/migrations/0000*00*00_000000_CreateUserTable.php** ファイルとして保存されます。ほとんどの 0 の部分は、他の数字に置き換わっているでしょう。

ここでは、ID、ユーザー名、パスワード、作成日時、更新日時のフィールドを追加しました。タイムスタンプフィールドを追加する短縮形もありますが、私は明示的にフィールドを追加するのが好みです。全てのフィールドを nullable と指定したため、デフォルト値は null になります。

マイグレーションを巻き戻す (rollback) 時に実行される drop メソッドも追加しています。ユーザーテーブルが存在しているなら、drop します。

タイムスタンプの短縮形は**http://laravel.com/docs/schema#adding-columns**（日本語版[a]）で確認できます。

[a] http://laravel4.kore1server.com/docs/schema#adding-columns

このマイグレーションは、データベースドライバーだけを使用したい場合でも活用できます。しかし通常、モデルと初期値設定（シーディング）を含めた、大きな準備の一部なのです。

モデルを作成する

Laravel4 は、要求されているインターフェイスメソッドを全て実装した **User** モデルを提供しています。以下は多少変更していますが、ほぼ基本通りです。[3]

```
<?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 getReminderEmail()
    {
        return $this->email;
    }
}
```

これは **app/models/User.php** ファイルとして保存します。

[3] PHP のバージョンは 5.4 以降を想定してコードされています。5.4 から、配列の array(...) を [...] と書けるようになりました。他にも変更点があります。

$table="user" プロパティーを定義していることに注目してください。この名前は、マイグレーションで定義したテーブル名と同じです。

User モデルは、**Eloquent** を拡張し、モデルが認証で有効であるかを確認し、リマインダー操作を行うために、2つのインターフェイスを implements しています。インターフェイスは後ほど確認しますが、インターフェイスが要求するメソッドに注意を払っておくことは、重要です。

Laravel4 ではユーザーを特定するために、メールアドレスでもユーザー名でも使用できますが、**getAuthIdentifier()** がリターンするフィールドとは異なっている必要があります[4]。**UserInterface** インターフェイスではパスワードフィールドの名前を指定しますが、**getAuthPassword*()** メソッドをオーバーライド／変更し、変えることができます。

getReminderEmail() メソッドは、パスワードリセットメールをユーザーに送信するためのメールアドレスをリターンします。これは必須です。

もしも、Laravel4 が最初から準備している認証メカニズムを壊すのが怖くなければ、他のどんなカスタマイズでも User モデルで自由に行えます。

初期値設定クラス (Seeder) を作成する

Laravel4 はシーディング（初期値設定）システムを備えており、最初のマイグレーションの後に、データベースにレコードを追加するために使用できます。プロジェクトの初期ユーザーを追加するために、次のような初期値設定クラスを作成します。

```
1  <?php
2
3  class UserSeeder
4  extends DatabaseSeeder
5  {
6      public function run()
7      {
8          $users = [
9              [
10                 "username" => "christopher.pitt",
11                 "password" => Hash::make("7h3iMOST!53cu23"),
12                 "email"    => "chris@example.com"
13             ]
14         ];
15
16         foreach ($users as $user)
17         {
18             User::create($user);
```

[4] getAuthIdentifier メソッドは通常 ID を指すため、つまり ID とパスワード以外の項目で、ユーザーを一意に特定できるフィールドが利用できます。使い道は少ないですが、フィールドの組み合わせにより、一意になるように指定することも可能です。

```
        }
    }
}
```

これは **app/database/seeds/UserSeeder.php** ファイルとして保存します。

これを実行すると、私のアカウントがデータベースに追加されます。しかし、実行するには、メインの **DatabaseSeeder** クラスに追加する必要があります。

```
<?php

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

これは **app/database/seeds/DatabaseSeeder.php** ファイルとして保存します。

これで、**DatabaseSeeder** クラスが起動されると、私のアカウントをユーザーテーブルに初期設定します。もしあなたが、マイグレーション、モデル、有効なデータベース接続を用意済みであれば、以下のコマンドで全てを実行できます。

```
composer dump-autoload
php artisan migrate
php artisan db:seed
```

最初のコマンドで、私達が作成した全ての新しいクラスが、確実にオートロードされるようにします。2番目は、マイグレーションで指定されたテーブルを作成します。3番目は、ユーザーテーブルに初期値を設定します。

認証の設定

認証メカニズムの設定オプションは多くありませんが、カスタマイズ可能です。

```php
<?php

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

これは **app/config/auth.php** ファイルとして保存します。

この設定は全て重要で、内容は名前通りです。リセットメールの中身のビューは、“**email**” ⇒ “**email.request**” で指定し、リセットトークンの有効期限は、“**expire**” ⇒ **60** で指定しています。

特に注意を払ってもらいたい部分は、“**email**” ⇒ “**mail.request**” です。これは Laravel にデフォルトの **app/views/emails/auth/reminder.blade.php** の代わりに、**app/views/email/request.blade.php** を読み込むように指定しています。

プロバイダーにハードコードされている設定オプションからも、様々な利点が得られます。いくつかを後ほど取り上げましょう。

ログイン

信頼できるユーザーにアプリケーションを利用してもらうため、情報を入力するログインページを作成しましょう。情報が有効なものであれば、そのユーザーのプロファイルページへリダイレクトしましょう。

レイアウトビューを作成する

アプリケーションのページを作成する前に、レイアウトのマークアップとスタイルを全て抽象化してしまうのは、賢いやり方です。この目的のため、Blade テンプレートエンジンを使用し、様々な要素を含んだレイアウト（テンプレート）ビューを作成しましょう。

では、まずレイアウトビューを作成する必要があります。

```
<!DOCTYPE html>
<html lang=" en" >
    <head>
        <meta charset="UTF-8" />
        <link
            type="text/css"
            rel="stylesheet"
            href="/css/layout.css" />
        <title>
            Tutorial
        </title>
    </head>
    <body>
        @include("header")
        <div class="content">
            <div class="container">
                @yield("content")
            </div>
        </div>
        @include("footer")
    </body>
</html>
```

これは **app/views/layout.blade.php** ファイルとして保存します。

レイアウトビューは、ほぼ標準の HTML ですが、Blade 特定のタグを2つ使用しています。**@include()** タグは、**header** や **footer** のような文字列の名前のビューを views ディレクトリーから読み込むように Laravel へ指示します。

.blade.php 拡張子を省略していることに気づきましたか?Laravel が自動的に付け加えてくれます。そしてさらに、レイアウトビューへ結合したデーターを取り込んだ両ビューへ結合してくれます。

2つ目の Blade タグは **@yield()** です。このタグへセクション名を指定し、そのセクションに保存されたデーターを出力します。アプリケーションのビューは、このレイアウトビューを拡張します。それと同時に、レイアウトのマークアップの中に自身のマークアップを埋め込むために、**content**(内容)セクションを指定します。実際にどのようにセクションを指定するか、簡単に確認してください。

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

これは **app/views/header.blade.php** ファイルとして保存します。

ヘッダーは2つの Blade タグを含んでおり、指定した名前のセクションにマークアップを保存し、テンプレートの中でレンダーできるように、Blade に指示しています。

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

これは **app/views/footer.blade.php** ファイルとして保存します。

フッターでも同様に、名前を指定したセクションにラップしたマークアップが取り込まれ、テンプレートの中で即時レンダーされます。

皆さん、それらのインクルードファイルのセクションでマークアップをラップする必要があるのか、不思議に思っていることでしょう。後ほど全て、同時にレンダーします。これにより、内容を変更することができるようになります。この後すぐ、お見せいたしましょう。

```
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, .error
{
    clear  : both;
    float  : left;
    margin : 5px 0;
}
.error
{
    color : #ef7c61;
}
```

これは **public/css/layout.css** ファイルとして保存します。

head 要素の中でリンクする、基本的なスタイルを付け加えておしまいです。デフォルトのフォントとレイアウトを変更するものです。アプリケーションは、これが無くても動作しますが、見かけが多少ごちゃごちゃになります。

ログインビューの作成

ログインビューは主にフォームです。ユーザーがログイン情報を入力するためのものです。

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::old("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "□□□□□□□□□□□"
        ]) }}
        {{ Form::submit("login") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

これは **app/views/user/login.blade.php** ファイルとして保存します。

ログインビューの最初の Blade タグは、Laravel にこのビューはレイアウトビューを拡張していることを伝えています。2つ目はマークアップがコンテンツ (content) セクションに含まれることを伝えています。レイアウトを除いて、これらのタグはこれから作成するビューにおいても、基本的に同様な形態で使用します。

次に {{と}} を使い、挟まれているコードを PHP として解釈するように、Laravel へ指示しています。**Form::open()** メソッドでフォームをオープンしています。最初の引数でポストするルートを指定し、2つ目の引数は省略可能ですが、パラメーターを渡しています。

それから、2つのラベルと、3つの入力を定義しています。ラベル (label) は名前、続いて表示テキストを引数に取ります。テキスト (text) 入力は、名前、デフォルト値、それから追加のパラメーターを引数に取ります。パスワード (password) 入力は、名前と追加のパラメーターです。最後に送信 (submit) 入力は、ラベルと同様に、名前と表示テキストです。

フォームを閉じるために、**Form::close()** を呼び出します。

> Laravel が提供している他の **Form** メソッドは、**http://laravel.com/docs/html**（日本語[a]）をご覧ください。
>
> [a] http://laravel4.kore1server.com/docs/html

ログインビューの最後で、（既に作成し、読み込んだフッターの中に指定されている）フッターのデフォルトマークアップをオーバーライドしています。同じ名前を使用していますが、**@show** でセクションを終えていません。取り込んだビューで定義しているため、既にレンダーされているでしょう。ですから、content セクションの最後に **@stop** を使用します。

さらに、**@parent** Blade タグを用い、デフォルトのフッターで定義されたマークアップを表示したいことを Laravel に伝えています。完全に変更するのではなく、スクリプトタグを追加するだけです。

> Blade タグの詳細は**http://laravel.com/docs/templates#blade-templating**（日本語[a]）でご覧ください。
>
> [a] http://laravel4.kore1server.com/docs/templates#blade-templating

取り込んでいるスクリプトは polyfill.io という名前です。ブラウザの違いを埋めるスクリプトのコレクションで、例えば **placeholder** 属性を使用できるようにします。（通常、古いブラウザーでは表示されません。）

> Polyfill.io については、**https://github.com/jonathantneal/polyfill**を参照してください。

これでログインビューは完璧です。しかし、入力を受け取り、結果を返す、サーバーサイドのコードがまだです。では、解決しましょう！

ログインのアクションを作成する

ログインアクションは、認証と既に作成したビューをくっつけるものです。ここまで一緒に作成していれば、いつになったらブラウザでここまでの作業の結果を目にできるのかと思っていたでしょう。ここまでのところ、アプリケーションにビューをロードするように、全く指示していません。

表示に取り掛かるには、ログインアクションのためにルートを追加する必要があります。

```
<?php

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

これは **app/routes.php** ファイルとして保存します。

新しくインストールしたての Laravel4 アプリケーションはデモページとして、routes ファイルから直接ビューをレンダーするように指示しています。これをコントローラー／アクションを使用するように変更する必要があります。必ずコントローラーを使う必要はありません。ロジックをルートファイルの中で実行するほうが簡単です。しかし、余り美しい方法ではありません。

“as” ⇒ “user/login” により、ルートに名前をつけています。行き先を **“uses” ⇒ “UserController@loginAction”** で指定しています。両方共にデフォルトルートの”/” に結び付けられています。そして名前により、簡単にこのルートを指し示すことができるようになります。

次に、コントローラーを作成する必要があります。

```
<?php

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

これは **app/controllers/UserController.php** として、セーブしてください。

Controller クラスを拡張し、**UserController** を定義しました。ルートファイルの中で指定した、**loginAction()** メソッドのみがあります。今のところ、ログインビューをブラウザーにレンダーするだけです。しかし、進捗を目で確認するには十分です！

ユーザー認証

そうです、フォームがあります。今度はデータベースにとりかかる必要があり、その結果、ユーザーが正しい人か認証できるようになります。

```
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                echo "Validation passed!";
            }
            else
            {
                echo "Validation failed!";
            }
        }

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

これは **app/controllers/UserController.php** として、セーブしてください。

UserController を変更しました。**loginAction** メソッドにポストされたデーターを操作する必要がありますが、その前にサーバーのプロパティ、**REQUEST_METHOD** を調べています。もし、この値が **POST** なら、フォームがこのアクションに対し送信されたものと仮定できます。それから続けて、バリデーションのフェーズに移ります。

> 同じページに対して、GET と POST を分ける手法もよく見受けられます。それにより、多少きれいになり、**REQUEST_METHOD** を調べる必要も無くなります。私は両方を同じアクションで処理するほうが好みです。

Laravel4 は素晴らしいバリデーションシステムを提供しています。それを使用する一つの方法は、**Validator::make()** を呼び出すことです。最初の引数は、バリデーションしたいデーターの配列、2つ目の引数はルールの配列です。

username と password フィールドに required を指定しただけですが、他にも多くのバリデーションルールが存在します。後ほど、そのうちのいくつかを使用します。**Validator** クラスは、**passes()** メソッドも持ち、ポストされたフォームデーターが有効であるかどうかを確認します。

> 場合により、バリデーションロジックをコントローラーの外部においたほうが良いでしょう。私はよくモデルの中に設置しますが、入力のバリデーションを処理するためのクラスを作成する手もあります。

このフォームをポストすると、必須のフィールドに入力されているか、いないかを表示する必要が出ます。この種のメッセージを表示するためのエレガントな方法が存在しています。

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $data = [];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                //
            }
            else
```

```
24              {
25                  $data["errors"] = new MessageBag([
26                      "password" => [
27                          "ユーザー／パスワードが無効です。"
28                      ]
29                  ]);
30              }
31          }
32
33          return View::make("user/login", $data);
34      }
35  }
```

これは **app/controllers/UserController.php** として、セーブしてください。

上の修正では、バリデーションエラーメッセージを保存するために、**MessageBag** クラスを使用しています。これはバリデーションクラスが暗黙の内にエラーを保存する方法と似ていますが、username と password それぞれのエラーメッセージを表示する代わりに、両方に対し一つのエラーメッセージを表示します。この方法でログインフォームは、多少安全性が増します！

このエラーメッセージを表示するために、ログインビューを変更する必要があります。

```
1   @extends("layout")
2   @section("content")
3       {{ Form::open([
4           "route"        => "user/login",
5           "autocomplete" => "off"
6       ]) }}
7           {{ Form::label("username", "Username") }}
8           {{ Form::text("username", Input::get("username"), [
9               "placeholder" => "john.smith"
10          ]) }}
11          {{ Form::label("password", "Password") }}
12          {{ Form::password("password", [
13              "placeholder" => "••••••••••"
14          ]) }}
15          @if ($error = $errors->first("password"))
16              <div class="error">
17                  {{ $error }}
18              </div>
19          @endif
20          {{ Form::submit("login") }}
21      {{ Form::close() }}
```

```
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

これは **app/views/user/login.blade.php** ファイルとして保存します。

ご覧の通り、エラーメッセージが存在しているかをチェックし、div 要素の中でレンダーしています。もし、バリデーションに失敗したら、パスワードの下にエラーメッセージが表示されるようになります。

入力と一緒にリダイレクトする

フォームに共通する落とし穴は、は、ページを更新すると、フォームが再送信されてしまうことでしょう。Laravel マジックでこれに打ち勝つことができます。ポストされたフォームのデーターをセッションに保存し、ログインページにリダイレクトで戻します！

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
```

```
25                  "password" => "required"
26              ]);
27
28              if ($validator->passes())
29              {
30                  //
31              }
32              else
33              {
34                  $data["errors"] = new MessageBag([
35                      "password" => [
36                          "ユーザー／パスワードが無効です。"
37                      ]
38                  ]);
39
40                  $data["username"] = Input::get("username");
41
42                  return Redirect::route("user/login")
43                      ->withInput($data);
44              }
45          }
46
47          return View::make("user/login", $data);
48      }
49  }
```

これは **app/controllers/UserController.php** として、セーブしてください。

最初に新しい **MessagaeMag** インスタンスを宣言しています。その理由は、セッションに保存されている、いないに関わらず、ビューで errors の **MessageMag** をチェックしているからです。しかし、セッションの中に存在していれば、新しく作成したインスタンスをオーバーライドします。

それからビューに渡し、レンダーできるように **$data** 配列に追加します。

バリデーションに失敗したら、バリデーションエラーと一緒に、username を **$data** 配列へ保存し、同じルートへリダイレクトします。（もしくは、データーをセッションに保存する、**withInput()** メソッドも使用できます。）

ビューは変更されないままですが、ひどいフォームの再送信を起こさず、更新できます。（おまけに、うるさいブラウザーメッセージが表示されるようになりました。）

認証情報

認証の最後のステップは、提供されたフォーム情報でデータベースを確認することです。Laravel は簡単に実現してくれます。

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                $credentials = [
                    "username" => Input::get("username"),
                    "password" => Input::get("password")
                ];

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

            $data["errors"] = new MessageBag([
```

```
42                  "password" => [
43                      "ユーザー／パスワードが無効です。"
44                  ]
45              ]);
46
47              $data["username"] = Input::get("username");
48
49              return Redirect::route("user/login")
50                  ->withInput($data);
51          }
52
53          return View::make("user/login", $data);
54      }
55  }
```

これは **app/controllers/UserController.php** として、セーブしてください。

単にポストされたフォーム情報 (**$credentials**) を **Auth::attempt()** メソッドへ渡すだけで、それが有効なものであるなら、そのユーザーをログインさせます。有効ならば、ユーザーのプロファイラーページへのリダイレクトをリターンします。

通常なら else で処理するエラー処理コードをも省いています。これによりバリデーションエラーと同時に、認証エラーを同じコードで処理できます。ログインページの場合、これで構いません。

パスワードリセット

Laravel4 に組み込まれている、パスワードリセットメカニズムは素晴らしいものです！ユーザーが自分のメールアドレスを指定することで、パスワードをリセットできるようにしてくれます！

パスワードリセットビューを作成する

パスワードのリセット機能を提供するためには、ユーザーに 2 つのビューを提供する必要があります。リセットトークンを送るためのメールアドレスを入力してもらうビューと、アカウントの新しいパスワードを指定してもらうビューです。

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/request",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

これは **app/view/views/user/request.blade.php** としてセーブします。

このビューはログインビューと似ており、項目がメールアドレスだけになっています。

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::route("user/reset") . $token,
        "autocomplete" => "off"
    ]) }}
        @if ($error = $errors->first("token"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        @if ($error = $errors->first("email"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password", "Password") }}
```

```
        {{ Form::password("password", [
            "placeholder" => "••••••••••"
        ]) }}
        @if ($error = $errors->first("password"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password_confirmation", "Confirm") }}
        {{ Form::password("password_confirmation", [
            "placeholder" => "••••••••••"
        ]) }}
        @if ($error = $errors->first("password_confirmation"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

これは **app/views/user/reset.blade.php** としてセーブします。

そうですね。皆さんここまでで、入力とエラーメッセージをフォームで取り扱う方法は学んでいます。注目してもらいたい重要な変更は、フォームのアクションです。ビューに渡す変数といっしょに、**URL::route()** を使用していることです。これをアクションの中でセットしていますが、今のところ心配しなくても大丈夫です。

パスワードトークンをリクエストするメールも少々変更していますが、Laravel4 をインストール時に最初から提供されている、デフォルトのビューとほとんど同じです。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Password Reset</h1>
        To reset your password, complete this form:
        {{ URL::route("user/reset") . "?token=" . $token }}
    </body>
</html>
```

これは **app/views/email/request.blade.php** ファイルとして保存します。

このビューは、デフォルトで **app/views/emails/auth/reminder.blade.php** ですが、設定オプションの emailing で変更していることを思い出してください。

パスワードリセットアクションを作成する

アクションへアクセスできるようにするため、ルートを追加しましょう。

```
<?php

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

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

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

これは **app/routes.php** ファイルとして保存します。

request ルートは、リセットトークが必要なこと、reset ルートでパスワードをリセットしていることを覚えておいてください。

パスワードリセットトークンテーブルを生成する必要もありますね。Artisan を使用しましょう。

```
php artisan auth:reminders
```

これで、リマインダーテーブルのマイグレーションテンプレートが生成されます。

```
<?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")
                ->nullable()
                ->default(null);

            $table
                ->string("token")
                ->nullable()
                ->default(null);

            $table
                ->timestamp("created_at")
                ->nullable()
                ->default(null);
        });
    }

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

```
33      }
34  }
```

これは **app/database/migrations/0000_00_00_000000_CreateTokenTable.php** ファイルとして保存されます。ほとんどの 0 は他の数字と置き換わっていることでしょう。

テンプレートを多少変更しました。基本的な部分は全部同じです。これにより認証メカニズムがパスワードリセットトークンを生成したり、バリデートするために使用される **email**、**token**、**created_at** フィールドを持ったテーブルが作成されます。

それでは、パスワードリセットアクションを追加しましょう。

```
1   public function requestAction()
2   {
3       $data = [
4           "requested" => Input::old("requested")
5       ];
6
7       if (Input::server("REQUEST_METHOD") == "POST")
8       {
9           $validator = Validator::make(Input::all(), [
10              "email" => "required"
11          ]);
12
13          if ($validator->passes())
14          {
15              $credentials = [
16                  "email" => Input::get("email")
17              ];
18
19              Password::remind($credentials,
20                  function($message, $user)
21                  {
22                      $message->from("chris@example.com");
23                  }
24              );
25
26              $data["requested"] = true;
27
28              return Redirect::route("user/request")
29                  ->withInput($data);
30          }
31      }
```

```
32
33      return View::make("user/request", $data);
34  }
```

これは、**app/controllers/UserController.php** の一部です。

requestAction() メソッドは、**loginAction()** メソッドと同様に、ポストされたフォームデーターをバリデートしますが、**Auth::attempt()** にフォームデーターを渡す代わりに、**Password::remind()** へ渡しています。このメソッドはユーザー情報（通常はメールアドレスだけが含まれます）の配列と、送信するメールをカスタマイズするためのコールバックをオプションとして、引数に取ります。

```
1   public function resetAction()
2   {
3       $token = "?token=" . Input::get("token");
4
5       $errors = new MessageBag();
6
7       if ($old = Input::old("errors"))
8       {
9           $errors = $old;
10      }
11
12      $data = [
13          "token"  => $token,
14          "errors" => $errors
15      ];
16
17      if (Input::server("REQUEST_METHOD") == "POST")
18      {
19          $validator = Validator::make(Input::all(), [
20              "email"                 => "required|email",
21              "password"              => "required|min:6",
22              "password_confirmation" => "same:password",
23              "token"                 => "exists:token,token"
24          ]);
25
26          if ($validator->passes())
27          {
28              $credentials = [
29                  "email" => Input::get("email")
30              ];
31
```

```
32              Password::reset($credentials,
33                  function($user, $password)
34                  {
35                      $user->password = Hash::make($password);
36                      $user->save();
37
38                      Auth::login($user);
39                      return Redirect::route("user/profile");
40                  }
41              );
42          }
43
44          $data["email"] = Input::get("email");
45
46          $data["errors"] = $validator->errors();
47
48          return Redirect::to(URL::route("user/reset") . $token)
49              ->withInput($data);
50      }
51
52      return View::make("user/reset", $data);
53  }
```

これは、**app/controllers/UserController.php** の一部です。

resetAction() メソッドはほとんど同じです。全リセットページでトークンが失われないようにするため、リダイレクトに使用する、トークンのクエリー文字列を最初に生成します。ログインページで行ったのと同じように、前回のエラーメッセージを取得し、ポストされたフォームデーターをバリデートします。

全てのデーターが有効なものであれば、**Password::reset()** へ渡します。2 番めの引数は、ユーザーのデータベースレコードを更新するためのロジックを指定するために使用します。パスワードを更新し、レコードを保存、それからそのユーザーを自動的にログインさせています。

何も引っかからなければ、プロファイルページへリダイレクトしています。引っかかれば、エラーメッセージといっしょに、リセットページへ戻るようにリダイレクトします。

> ここで認証メカニズムには、何だか不可解な部分もあることをお話しましょう。password と token フィールド名がハードコードされており、**Passwrod::reset()** メソッドの中でハードコードされたバリデーションが組み込まれていますが、**Validation** クラスを使用していません。そのためフィールド名に **password**、**password_confirmation**、**token** を使っている限り、パスワードは 6 文字以上必要なのです。この風変わりな仕様に皆さん気がついていないでしょう。
>
> 代わりに、フィールド名と、**vendor/laravel/framework/src/Illuminate/Auth/Reminders/-**

PasswordBroker.php ファイルの中のバリデーションを変更するか、Larvel4 が提供している **ReminderServiceProvider** を自分で実装したものへ変更することもできます。両方のアプローチの詳細は、このチュートリアルの範疇を超えています。サービスプロバイダーを作成する詳細については、Taylor Otwell 氏の素晴らしい書籍、**https://leanpub.com/laravel**（日本語版[a]）をご覧ください。

[a] https://leanpub.com/laravel-jp

既に述べた通り、**app/config/auth.php** ファイルで、パスワードトークンの有効期限時間をセットすることができます。

認証のメソッドについての詳細は、**http://laravel.com/docs/security#authenticating-users**（日本語版[a]）をご覧ください。

[a] http://laravel4.kore1server.com/docs/security#authenticating-users

メール送信のメソッドについては、**http://laravel.com/docs/mail**をご覧ください。

ユーザー認証に取り掛かる

はい。ここまでで、ログインとパスワードリセットを実装しました。このチュートリアルの最後は、アプリケーションのユーザーセッションデーターを使用すること、それとアプリケーションの秘密のページへの認証されていないアクセスを防ぐことです。

プロファイルページを作成する

ユーザーセッションデーターへのアクセスを示すため、プロファイルビューを実装しましょう。

```
1 @extends("layout")
2 @section("content")
3     <h2>いらっしゃい、{{ Auth::user()->username }}</h2>
4     <p>中身のないプロファイルページへようこそ。</p>
5 @stop
```

これは、**app/views/user/profile.blade.php** ファイルとして保存します。

この、驚異的に内容が何もないプロファイルページが伝えているのは、たった一つのことです。**Auth::user()** により返されるアクセスオブジェクトを使用し、ユーザーモデルからデーターを取得できることです。モデルに定義した（つまりデータベーステーブルのフィールド）へ、この方法でアクセス可能です。

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

これは、**app/controllers/UserController.php** の一部です。

profileAction() メソッドは、ビューと同様にシンプルです。ビューに何もデーターを渡す必要はありません。特別なコードでユーザーのセッションを保持しておく必要もありません。**Auth::user()** が全て行います！

このページにアクセスできるように、ルートを追加する必要がありますね。すぐ後にやります。しかし、アプリケーションの機密ページを守ることについてお話する良い時期がきたようです…

フィルターを作成する

Laravel4 にはフィルターファイルが存在しており、ある一つのルート（もしくはグループ）に対するフィルターを定義できます。

```
<?php

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

Route::filter("guest", function()
{
    if (Auth::check())
    {
```

```
15          return Redirect::route("user/profile");
16      }
17  });
18
19  Route::filter("csrf", function()
20  {
21      if (Session::token() != Input::get("_token"))
22      {
23          throw new Illuminate\Session\TokenMismatchException;
24      }
25  });
```

これは **app/filters.php** ファイルとして保存します。

最初はユーザー認証が必要なルート(お好みでしたらページと言い換えてください)のためのフィルターです。2つ目は真逆で、認証されていないユーザーのためのものです。最後のフィルターは、本当なら最初から使うべきだったものです。

Form::open() メソッドを使用すると、Laravel は自動的にフォームに隠しフィールドを追加します。このフィールドは特別なセキュリティートークンで、フォームが送信されるたびにチェックされます。あなたはこれにより、なぜもっと安全になるかを理解する必要は本当にないのですが…

…しかし、知りたければ、**http://blog.ircmaxell.com/2013/02/preventing-csrf-attacks.html**を読んでください。

このフィルターを適用するように、ルートファイルを書き換えましょう。

```
1   <?php
2
3   Route::group(["before" => "guest"], function()
4   {
5       Route::any("/", [
6           "as"   => "user/login",
7           "uses" => "UserController@loginAction"
8       ]);
9
10      Route::any("/request", [
11          "as"   => "user/request",
12          "uses" => "UserController@requestAction"
13      ]);
```

```
14
15      Route::any("/reset", [
16          "as"   => "user/reset",
17          "uses" => "UserController@resetAction"
18      ]);
19  });
20
21  Route::group(["before" => "auth"], function()
22  {
23      Route::any("/profile", [
24          "as"   => "user/profile",
25          "uses" => "UserController@profileAction"
26      ]);
27
28      Route::any("/logout", [
29          "as"   => "user/logout",
30          "uses" => "UserController@logoutAction"
31      ]);
32  });
```

これは **app/routes.php** ファイルとして保存します。

アプリケーションの一部を守るために、**Route::group()** メソッドを使用してグループをひとまとめに取り扱っています。最初の引数はクロージャーの中のルートに対して適用するフィルターです。ログインしているユーザーには見せないように、認証されているユーザーが入ってはいけない全てのルートをまとめています。それとは逆に、プロファイルページは、認証済みのユーザーに対してのみ、見せる必要があるのです。

ログアウトアクションを作成する

これらのセキュリティー機能を確かめるため、（それと、このチュートリアルを仕上げてしまうためにも）ユーザーがログアウトできるように、**logoutAction()** メソッドを作成し、ヘッダーへリンクを追加しましょう。

```
1  public function logoutAction()
2  {
3      Auth::logout();
4      return Redirect::route("user/login");
5  }
```

これは、**app/controllers/UserController.php** の一部です。

logoutAction() メソッドは、ユーザーセッションをクローズするために **Auth::logout()** メソッドを呼び出し、それからログインページへリダイレクトしています。本当に簡単です！

新しいヘッダーは、この様になります。

```
@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
```

これは **app/views/header.blade.php** ファイルとして保存します。