LARAVEL IN PRATICA

UNA GUIDA VELOCE PER REALIZZARE SITI WEB DI QUALITA'

di Francesco Lettera

Laravel 4!

Laravel in pratica

Una guida veloce per realizzare siti web di qualità

Francesco Lettera

Questo libro è in vendita presso http://leanpub.com/laravel_in_pratica

Questa versione è stata pubblicata il 2017-10-28

Twitta questo libro!

Per favore, aiuta Francesco Lettera diffondendo la parola a proposito di questo libro su Twitter!

Il tweet consigliato per questo libro è:

Ho appena comprato Laravel in Pratica

L'hashtag suggerito per questo libro è #laravel.

Scopri quello che gli altri hanno da dire sul libro cliccando su questo link per cercare questo hashtag su Twitter:

#laravel

Indice

Capitolo 3: l'approccio RESTful

Gestiamo le categorie

La tabella delle categorie sarà l'anello di congiunzione tra gli utenti della nostra intranet e i documenti caricati dall'admin. Occupiamoci della tabella utilizzando il migration system di Laravel che già conosciamo. Posizioniamoci con il prompt dei comandi in `C:\wamp\www` e digitiamo:

```
php artisan migrate:make create_categorie --create=categorie
```

Modifichiamo il metodo `up()`, come di consueto.

```
1  // app/database/migrations/2013_07_29_143818_create_categorie_table.php
2
3      public function up()
4      {
5          Schema::create('categorie', function(Blueprint $table)
6          {
7              $table->increments('id');
8              $table->string('nome_categoria');
9              $table->timestamps();
10         });
11     }
```

Poi ritorniamo sul nostro prompt di comandi e digitiamo:

```
php artisan migrate
```

Nel nostro db sarà quindi presente la nostra tabella categoria, tutta vuota e pronta per essere riempita.

Un approccio RESTful per le categorie

Gestire le risorse seguendo un approccio RESTful ci garantisce una serie di regole standard. Tali regole saranno applicate per quasi tutte le parti del nostro progetto. Nel corso del libro, infatti, gestiremo anche gli utenti con questo approccio, e le risorse. Laravel ci viene in aiuto perché ha già concepito l'approccio RESTful per i suoi Controller. Dal nostro insostituibile prompt dei comandi digitiamo (siamo sempre in `C:\wamp\www`):

```
php artisan controller:make CategorieController
```

E' stato appena creato un nuovo file nella cartella `app/controllers`

```php
class CategorieController extends \BaseController {

    /**
     * Display a listing of the resource.
     *
     * @return Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @return Response
     */
    public function store()
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        //
    }
```

```
43
44      /**
45       * Show the form for editing the specified resource.
46       *
47       * @param  int  $id
48       * @return Response
49       */
50      public function edit($id)
51      {
52          //
53      }
54
55      /**
56       * Update the specified resource in storage.
57       *
58       * @param  int  $id
59       * @return Response
60       */
61      public function update($id)
62      {
63          //
64      }
65
66      /**
67       * Remove the specified resource from storage.
68       *
69       * @param  int  $id
70       * @return Response
71       */
72      public function destroy($id)
73      {
74          //
75      }
76
77  }
```

Wow! Quanta roba. I metodi sono intuitivi, del resto sono ben commentati. Nulla ci vietava di creare questo controller a mano, ma la forza di `artisan` è proprio quella di abbreviare alcuni aspetti comuni della programmazione PHP. Un paio di ulteriori accorgimenti prima di procedere: creiamo un file `Categorie.php` all'interno della cartella `model`.

```php
// app/models/Categorie.php

class Categorie extends Eloquent {

    protected $table = 'categorie';

}
```

Ho creato una classe che estende Eloquent. La utilizzeremo per le nostre query richiamate dal Controller `CategorieController.php`.

Poi passiamo al capitano di rotta (`routes.php`) e indichiamogli che desideriamo utilizzare un approccio RESTful:

```php
// app/routes.php

Route::resource('categorie', 'CategorieController');
```

Questa singola dichiarazione crea una serie di regole di route che gestiranno i metodi RESTful del nostro controller. Diamo un'occhiata a questa tabella:

METODO	PERCORSO	AZIONE	NOME ROUTE
GET	/categorie	index	categorie.index
GET	/categorie/create	index	categorie.create
POST	/categorie	store	categorie.store
GET	/categorie/{id}	show	categorie.show
GET	/categorie/{id}/edit	edit	categorie.edit
PUT/PATCH	/categorie/{id}	update	categorie.update
DELETE	/categorie/{id}	destroy	categorie.destroy

La tabella elenca tutti gli stati del controller RESTful. Avrai già notato che il percorso `index` è simile al percorso `store`. In realtà la differenza sostanziale è nel metodo: il primo è GET, il secondo è POST. Cioè Laravel riconosce se i dati provengono tramite GET o POST e reindirizza al metodo opportuno. Ma diamo vita a questo controller.

Inserimento della categoria

Prima di tutto il form per inserire la categoria. Andrà nel metodo `create()`:

```
// app/controllers/CategorieController.php

public function create()
{
    $this->layout->content = View::make('categorie.create');
}
```

Poi creiamo la nostra view. Già sappiamo come fare:

```
// app/views/categorie/create.blade.php

@section('content')

{{ Form::open(array('url' => 'categorie', 'method' => 'POST')) }}
<div class="row">
    <div class="col-lg-3">
        <div class="form-group">
        {{ Form::label('nome_categoria', 'Nome categoria') }}
        {{ Form::text('nome_categoria', '', array('class'=>'form-control')) }}
        </div>
    </div>
</div>

<div class="row">
    <div class="col-lg-3">
        <div class="form-group">
            {{ Form::submit('Aggiungi questa categoria',  array('class' =>'btn b\
tn-success btn-large')) }}
        </div>
    </div>
</div>
{{ Form::close() }}
@stop
```

Diamo un'occhiata al parametro `url` del form: punta a `categorie` e il suo metodo è `POST`. Se diamo un'occhiata alla tabella di prima, sapremo già quale metodo recupererà il campo `nome_categoria`: è il metodo `store()`. Occupiamoci di quest'ultimo:

```php
// app/controllers/CategorieController.php

public function store()
{
    $categorie = new Categorie;
    $categorie->nome_categoria = Input::get('nome_categoria');
    $categorie->save();
    return Redirect::action('CategorieController@index');
}
```

Avrai notato che ho istanziato l'oggetto `categorie` per utilzzarlo nel metodo. `$categorie->nome_categoria` è infatti il nome del campo della tabella `categorie`: l'ho valorizzato con l'input recuperato dal form (noterai l'orrore della mancanza di validazione. Don't worry, ne parleremo più in là) e poi ho dato il comando `save()` (E qui noterai l'eleganza di Laravel!) per salvare il tutto nel db. Fatto! Dopo questa operazione, un redirect al metodo `index()` del nostro controller. A proposito, quando dobbiamo redirezionare verso un metodo del controller, il metodo (perdona il bisticcio) è `action`:

```php
 return Redirect::action('CategorieController@index');
```

Elenco delle categorie

Realizziamo adesso la lista dei dati inseriti. Questa lista sarà all'interno del metodo `index()` del nostro controller.

```php
// app/controllers/CategorieController.php

public function index()
{
    $data['categorie_lista'] = Categorie::all();
    $this->layout->content = View::make('categorie.categorie_lista', $data);
}
```

Con `Categorie::all()` recuperiamo immediatamente la lista delle categorie inserite. Si tratta di una query banale, e sarebbe antieconomico realizzare un metodo all'interno del model `Categorie.php`. Laravel ci strizza l'occhio e ci dà subito la lista. Per query più complesse, però, è sempre consigliabile utilizzare i metodi nel model. Anche per una questione di ordine. Passiamo dunque l'array valorizzato `$data['categorie_lista']` alla view:

```php
    $this->layout->content = View::make('categorie.categorie_lista', $data);
```

E realizziamo la view. Ho pensato ad una gloriosa tabella:

```
// app/views/categorie/categoria_lista.blade.php

@section('content')
<table class="table">
    <tr>
        <td>Categoria</td>
        <td>Azioni</td>
    </tr>
@foreach($categorie_lista as $c)
    <tr>
        <td>{{ $c['nome_categoria'] }}</td>
        <td><a href="{{ url('categorie/'.$c['id'].'/edit') }}" class="btn btn-wa\
rning">Modifica</a></td>
        <td><a href="{{ url('categorie/'.$c['id']) }}" class="btn btn-warning">C\
ancella</a></td>
    </tr>
@endforeach
</table>

@stop
```

Se hai letto l'introduzione a Blade non hai bisogno di ulteriori spiegazioni sulla sintassi di questa view. Vorrei, però, farti notare i due link della colonna AZIONI: il primo serve per la modifica e punta a `categorie/{id}/edit`. Cioè al metodo `edit()` (dài un'occhiata alla tabella: tutto è come previsto); il secondo link, invece, punta a `categorie/{id}` con metodo GET, quindi sarà accolto da `show()`.

Recupero e modifica della categoria

Il recupero della categoria da modificare è affidato al metodo `edit()`:

```
// app/controllers/CategorieController.php
    public function edit($id)
    {
        $data['categoria_dettaglio'] = Categorie::find($id);
        $this->layout->content = View::make('categorie.edit', $data);
    }
```

Anche in questo caso, come l'elenco completo delle categoria, la query è piuttosto semplice ed Eloquent ci soccorre con un'istruzione più che mai chiara: `find()`. Ecco la nostra view:

```
// app/views/categorie/edit.blade.php

@section('content')

{{ Form::open(array('url' => 'categorie/'. $categoria_dettaglio->id, 'method' =>\
 'PUT')) }}
<div class="row">
    <div class="col-lg-3">
        <div class="form-group">
        {{ Form::label('nome_categoria', 'Nome categoria') }}
        {{ Form::text('nome_categoria', $categoria_dettaglio->nome_categoria, ar\
ray('class'=>'form-control')) }}
        </div>
    </div>
</div>

<div class="row">
    <div class="col-lg-3">
        <div class="form-group">
            {{ Form::submit('Aggiorna questa categoria',  array('class' =>'btn b\
tn-success btn-large')) }}
        </div>
    </div>
</div>
{{ Form::close() }}
@stop
```

Piuttosto simile alla view `categorie/create.blade.php`, ma ti invito a dare un'occhiata all'url del form e al metodo. Il primo punta a `categorie/{id}` e il metodo è `PUT`. Quindi, da tabella, punterà a `update()`:

```
 // app/controllers/CategorieController.php

    public function update($id)
    {
         $categoria = Categorie::find($id);
         $categoria->nome_categoria = Input::get('nome_categoria');
         $categoria->save();
         return Redirect::action('CategorieController@index');
    }
```

Anche in questo caso utilizziamo `find($id)` come nel metodo `edit()`, ma il fine è diverso. Abbiamo infatti recuperato il record, sostituito con il nuovo `input` e salvato nel db con il comando che già conosciamo `save()`.

Cancellazione della categoria

La view che mostrava la lista delle categoria conteneva anche il link per la cancellazione. Eccolo qui:

```
// app/views/categorie/categoria_lista.blade.php

 <td><a href="{{ url('categorie/'.$c['id']) }}" class="btn btn-warning">Cancella\
</a></td>
```

Se riprendiamo ancora una volta la tabella di tutti i metodi RESTful, noteremo che questo URL corrisponde a:

METODO	PERCORSO	AZIONE	NOME ROUTE
GET	/categorie/{id}	show	categorie.show

Dunque il metodo è `show()`:

```
 // app/controllers/CategorieController.php

    public function show($id)
    {
        $data['categoria_dettaglio'] = Categorie::find($id);
        $this->layout->content = View::make('categorie.show', $data);
    }
```

Niente che tu non abbia già visto. Ma diamo un'occhiata alla view: è lei che farà la differenza:

```
// app/views/categorie/show.blade.php

@section('content')
<div class="row">
    <div class="col-lg-6">
        <div class="form-group">
        <h3>Sei sicuro di voler cancellare questa categoria?</h3>
        </div>
    </div>
</div>
```

```
11
12 {{ Form::open(array('url' => 'categorie/'. $categoria_dettaglio->id, 'method' =>\
13  'DELETE')) }}
14 <div class="row">
15     <div class="col-lg-3">
16         <div class="form-group">
17             {{ $categoria_dettaglio->nome_categoria }}
18         </div>
19     </div>
20 </div>
21 <div class="row">
22     <div class="col-lg-3">
23         <div class="form-group">
24             {{ Form::submit('Cancella questa categoria',  array('class' =>'btn b\
25 tn-success btn-large')) }}
26         </div>
27     </div>
28     <div class="col-lg-3">
29         <a href="{{ url('categorie') }}" class="btn btn-warning btn-large">No, c\
30 i ho ripensato</a>
31     </div>
32 </div>
33 {{ Form::close() }}
34 @stop
```

La particolarità è nel `method="DELETE"` del form. Ancora una volta la tabella:

METODO	PERCORSO	AZIONE	NOME ROUTE
DELETE	/categorie/{id}	destroy	categorie.destroy

Perfetto. Cliccando sul bottone `Cancella questa categoria` inneschiamo il metodo `destroy()` del nostro controller:

```
1 // app/controlers/CategorieController.hp
2
3     public function destroy($id)
4     {
5      $categoria = Categorie::find($id);
6      $categoria->delete();
7      return Redirect::action('CategorieController@index');
8     }
```

Ancora `find($id)` e poi `delete()`.

Un ritocco al template

E' arrivato il momento di ritoccare il nostro template per renderlo più carino. Aggiungiamo questa porzione di codice prima del `<div="container">`

```
// app/views/template/main.blade.php
      <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-\
target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Intranet Project</a>
        <div class="nav-collapse collapse">
          <ul class="nav navbar-nav">
            <li class="active"><a href="{{ url('/login') }}">Login</a></li>
          </ul>
        </div><!--/.nav-collapse -->
      </div>
    </div>
```

Poi, prima della chiusura del tag `</head>`, questa piccola regola CSS che aggiunge un po' di margine superiore:

```
<style type="text/css">
  body {
    padding-top: 70px;
  }
</style>
```

Il template corretto dovrebbe avere queste sembianze:

```
// app/views/template/main.blade.php

  <!DOCTYPE html>
<html>
  <head>
    <title>Bootstrap 101 Template</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- Bootstrap -->
    <link rel="stylesheet" href="{{ url('/bs/css/bootstrap.min.css') }}" media="\
screen">
    <style type="text/css">
      body {
        padding-top: 70px;
      }
      </style>
  </head>
  <body>
      <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-\
target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Intranet Project</a>
        <div class="nav-collapse collapse">
          <ul class="nav navbar-nav">
            <li class="active"><a href="{{ url('/login') }}">Login</a></li>
          </ul>
        </div><!--/.nav-collapse -->
      </div>
    </div>
    <div class="container">
      <div class="row">
        <div class="col-lg-12">
     @yield('content')
        </div>
      </div>
    </div>
    <!-- JavaScript plugins (requires jQuery) -->
    <script src="{{ url('http://code.jquery.com/jquery.js') }}"></script>
```

```
43      <!-- Include all compiled plugins (below), or include individual files as ne\
44  eded -->
45      <script src="{{ url('/bs/js/bootstrap.min.js') }}"></script>
46     </body>
47  </html>
```

Cosa abbiamo imparato

Un approccio RESTful offre una serie di operazioni standard per gestire risorse. Utilizzarlo garantisce una certa omogeneità, soprattutto quando si realizza un'applicazione backend.

Download dell'applicazione

Il progetto web fin qui discusso lo puoi scaricare a questo indirizzo https://www.dropbox.com/s/laptcpv1cnng58c/laravel_2.zip. Se desideri testare subito l'applicazione (saltando tutti gli step del libro), ricordati di lanciare il comando `php artisan migrate` per aggiornare il tuo db.