Linnéuniversitetet
Kalmar Växjö

Fördjupningsarbete
1DV450, Webbramverk

Mikael Östman, VT13

Abstrakt

Att skriva en serverapplikation i PHP med MVC-design kan bli ett omfattande och möjligtvis svårt jobb, då PHP i grunden inte har någon form av designmönster. Inte bara själva organisationen av alla applikationens filer kan vara ett huvudbry, utan även säkerheten för applikationer som kommunicerar mycket med en databas är något som ofta kräver mycket uppmärksamhet. Det finns flertalet ramverk för PHP som skapats för att kunna underlätta dessa saker för utvecklare, och i denna rapport kommer en av de senare och mycket lovande att undersökas, nämligen Laravel.

Bakgrund

Efter att tidigare skrivit databasdrivna PHP-applikationer med MVC-struktur, så vet jag hur det kan bli en hel del kod att organisera och hålla koll på. Även felhantering, speciellt för databas-delen, samt validering, kan vara något som kräver onödigt mycket arbete, istället för att kunna fokusera på att bygga funktionerna som behöver den funktionaliteten.

Därför har intresset för att använda ett ramverk för att bygga framtida applikationer uppstått, speciellt efter att ha provat på Ruby on Rails samt Django. Båda dessa ramverk fann jag enkla att snabbt sätta sig in i, i alla fall nog för att kunna börja bygga någon slags applikation. Dock så är stödet bland webbhotellen för båda dessa tekniker inte särskilt utspritt, speciellt inte bland svenska webbhotell.

Utifrån detta kommer intresset att använda ett ramverk för PHP, då detta är en av dom vanligare tekniker, och där stödet är mycket mer utspritt. Det är här Laravel kommer in i bilden. Det är ett ramverk för PHP, som siktar på att hjälpa utvecklare att komma över dessa brister, om man vill kalla det det, i språket, och anses av många som en värdig konkurrent till CodeIgniter, ett annat mycket populärt ramverk för PHP. Flera anser till och med att Laravel ska vara bättre än CodeIgniter, då de gått över till just Laravel.

Men hur lätt är det att komma igån med Laravel? Hur passa bra löser detta ramverk de problem som presenterats? Finns det ett bra community kring det? Detta är vad jag tänkt undersöka i denna rapport.

Metod

För att undersöka ramverket Laravel ska en enklare blogg byggas, föra att testa aspekter så som att arbeta mot en databas och ha ett inloggningssystem. Det applikationen ska klara av att göra, är att tillåta en administratör att publicera inlägg. Detta kommer att testa både autentisering samt databaskopplingen, vilka är delar som är vanliga bland många webbapplikationer.

De delar som kommer att testas genom detta, är hur kopplingen och kommunikationen mot en databas fungerar rent praktiskt för att kunna bygga en applikation, och hur implementationen av ett inloggningssystem kan genomföras.

Resultat

Grund för applikation

Att få en fungerande applikation, mappstruktur och filer, går väldigt fort och utan problem, då den arkivfil som hämtas från Laravels egen sida, redan innehållar alla mappar som behövs, så som mappar för modell, vy och kontroller. Det finns även redan en mycket simpel vy och kontroller redan implementerade, som ett slags exempel med kommentarer och instruktioner till hur man går tillväga för att börja skapa sin egen applikation.

Routing

Grunden till att kunna navigera i applikationen, sköts med hjälp av ramverkets routingsystem. Det är detta som bestämmer vilken del som ska köras, beroende på vilken adress som är aktiv. Exempelvis så har en applikation, en blogg som i detta fall, en del för besökare som kommer till sidan, som ska kunna visa allt det material som finns, och sedan även en administrationsdel, där administratören för sidan ska kunna sköta sidans innehåll.

Säg att adressen till startsidan är www.example.com. Det som routingen nu gör, är att istället för att ha någon lång och obskyr adress för att komma åt administrationsbiten till sidan, så går det att sätta www.example.com/admin till att vara kopplad till administrationssidan. Likadant så hjälper det till att skapa renare adresser för att bläddra bland det material som finns. Adressen www.example.com/post/154 skulle exempelvis vara en väldigt enkel och städad adress.

Så det routingen egentligen är till för, är att baserat på vad adressen innehåller, så ska olika funktioner köras och olika vyer visas. Detta sätt att jobba på i en applikation är ingenting som är nytt för just Laravel. Det som är lite mer speciellt med det system för routing som återfinns i Laravel, är att det antingen går att skicka vidare anrop till applikation antingen genom routes.php, den fil som innehåller information om applikationens routing, som man kan vara van vid från exempelvis Ruby on Rails eller Django. Eller, så kan man även bara välja att registrera sin kontroller i denna fil, och sedan istället styra detta i varje kontroller. Detta gör att det finns mycket flexibilitet, beroende på hur man själv föredrar att jobba med denna bit.

Modell

Grunden för all data i en applikation med MVC-struktur är modeller, som representerar den data som finns i databasen. Så först och främst behövs en databas med lämpliga tabeller för applikationen. I fallet för vår lilla applikation, så behövs en tabell för att spara användare i, för att kunna begränsa vem som kan logga in skriva inlägg, och en tabell för spara blogginlägg i. Den förstnämnda kan lämpligen döpas till *Users,* och en andra tabellen döps till *posts.*

För att underlätta skapandet av tabeller, så jobbar Laravel med något som kallas migrationsfiler. I dessa filer skriver man, på ett för ramverket specifikt sätt, hur man vill att tabellen sak se ut, vilka fält/kolumner den ska innehålla och vilka egenskaper tabellen ska ha. Detta kan underlätta då man själv kan välja mellan flera olika typer av databaser som ramverket har stöd för. I detta fall har jag valt att använda MySQL, men sättet att arbeta mot databasen, vilken typ ställts in, är likadant. Laravel har i sitt grundutförande stöd för MySQL, PostgreSQL, SQLite samt SQL Server.

Det Laravel sedan göra, utifrån instruktionerna i migrationsfilen, är att översätta dessa till instruktioner för den typ av databas som ställts in, i vårt fall MySQL. Men innan det går att göra detta, så måste först en speciell tabell skapas, som hjälper till att hålla koll på vilka migrationer som genomförts och vilka som ska genomföras. Allt detta görs med hjälp av artisan, ett kommandotolksverktyg som följer med Laravel. Genom att köra vissa kommandon, så hjälper detta verktyg till att skapa, i detta fallet, migrationsfiler och tabeller i databasen utifrån vad vi själva väljer att lägga i dessa filer. För att komma igång med migreringen, så ska kommandot `php artisan migrate:install` köras genom kommandotolken. Detta skapar en speciell tabell som hjälper Laravel att hålla koll på vilka migreringar som redan genomförts. För att sedan köra migreringarna, så körs `php artisan migrate`. Då körs alla de migrationsfiler som inte ännu körts.

Exempel,utdrag från migrationsfil för att skapa tabellen users:

```
Schema::create('users', function ($table) {
      $table->increments('id');
      $table->string('username', 128);
      $table->string('nickname', 128);
      $table->string('password', 64);
      $table->timestamps();
});
DB::table('users')->insert(array(
    'username'  => 'admin',
    'nickname'  => 'Admin',
    'password'  => Hash::make('password')
));
```

Exempel, utdrag från migrationsfil för att skapa tabellen posts:

```
Schema::create('posts', function ($table) {
      $table->increments('id');
      $table->string('title', 128);
      $table->text('content');
      $table->integer('author_id');
      $table->timestamps();
});
```

I exemplen ovan, så skapas först tabellen som ska innehålla användare(*users*), och sedan tabellen

som ska innehålla inläggen(*posts*). Det som är nämnvärt att gå igenom lite extra i dessa kodstycken, är raden `$table->timestamps();` som återfinns i båda tabellerna. Detta skapar två fält i databasen, *created_at* och *updated_at*. Det skapas även ett *author_id* i tabellen *posts*, vilken kommer innehålla ett id från tabellen *users,* för att kunna se vilken användare som skapade ett visst inlägg. Själva kopplingen mellan tabellen *users* och *posts* sköts i modellklassen. Vi skapar även en användare direkt när vi skapat tabellen *users*, för att underlätta genom att redan från början ha något att arbeta med.

När de nödvändiga tabellerna nu skapats med alla de nödvändiga fälten, så behövs även en modellfil som representerar respektive tabell som skapats. Som det tidigare nämndes, så finns det redan en mappstruktur från start. I denna struktur finns det en mapp speciellt för applikationens modeller, nämligen mappen *models*, där ramverket förväntar sig att hitta modeller tillhörande applikationen. I denna mapp skapar vi i en fil som heter *User.php* och en *Post.php*, och i respektive fil en klass som heter *User* samt *Post*, vilka båda ärver från *Eloquent*. Eloquent är Laravels sätt att arbete med objekt, för att underlätt kopplingen mellan databas och applikation, eller rättare sagt ett av sätten. Det finns även något som kallas *Fluent*, även detta ett sätt att hämta objekt och jobba med objekt från databasen. Sättet att jobba med *fluent* påminner något mer om att arbeta med sql-satser.

Exempel, hämta en post med Eloquent:

```
$post = Post::find($post_id);
```

Exempel, hämta en post med Fluent query builder:

```
$post = DB::table('posts')->find($post_id);
```

Själva modellfilerna i detta fall behöver inte innehålla särskilt mycket kod alls, då ramverket själv ser till att alla nödvändiga fält blir tillgängliga, tack vare Eloquent. Det ända som behövs läggas till i klasserna för detta exempel, är relationen mellan *User* och *Post*.

Exempel, modellen User:

```
class User extends Eloquent
{
	public function posts()
	{
		return $this->has_many('Post');
	}
}
```

Exempel, modellen Post:

```
class Post extends Eloquent
{
	public function author()
```

```
        {
                return $this->belongs_to('User', 'author_id');
        }
}
```

Som det syns i exemplen ovan, så lägger vi till en funktion på *User* som anger att en användare kan ha flera inlägg, vilket via Laravels sätt att sätta dessa relationer på, resulterar i väldigt lättläst kod. För *Post*, så skapar vi en funktion som anger att ett inlägg tillhör, ägs, av en användare. Denna relation kopplas till fältet *author_id*, som vi skapade tidigare. Funktionernas namn, *posts* och *author*, är döpta så för att senare kunna exempelvis hämta ut ett inläggs ägare, genom att skriva som nedan.

Exempel, hämta ägare från inlägg:

```
$post = Post::find($post_id);
$author = $post->author;
```

Variabeln *author* kommer i detta fallet innehålla ett Eloquent-objekt av typen *User*. Skulle vi istället vilja få fram alla inlägg som en viss användare skapat, skulle vi kunna skriva som nedan.

Exempel, hämta alla inlägg från en användare:

```
$user = User::find($user_id);
$posts = $user->posts();
```

Detta gör Eloquent till ett väldigt enkelt sätta att arbeta mot en databas på, utan att behöva spendera en massa tid på att själv bygga ett lager som har till uppgift att kommunicera med databasen.

Genom att skapa en *user*-tabell som denna, blir det sedan väldigt enkelt att implementera autentisering senare.

Som du säkert lagt märke till, är att tabellerna heter *users* samt *posts*, men modellerna heter *User* samt *Post*. Detta är ingen felskrivning, utan det är så att ramverkat tar namnet på modellen i pluralform, och letar sedan efter denna tabell i databasen. Detta är standardbeteendet, men det går även själv att specifiera vad tabellen heter, om dessa namn nu skulle skilja sig åt. Detta fungerar bäst med engelska ord, men det går att lägga till sina egna ord och regler för detta i en konfigurationsfil.

Controller

I kontrollern så bestäms det vad som ska göras, beroende på vad användaren gick till för adress. Detta jobb kan i Laravel antingen göras i kontrollern, eller direkt routing-filen. I detta exempel så sköter routingen endast dirigeringen till olika kontrollerfunktioner, då detta bättre följer MVC.

Redan från början finns det en mapp döpt till *controllers*, där ramverket kommer att leta efter applikationens kontrollerklasser, vilka alla ska ärva från *Base_Controller*. I dessa kontrollerklasser finns sedan en mängd olika funktioner. Dessa definierar man själv beroende på hur många sidor som kommer vara kopplade till just denna kontroller.

Det finns två sätt att skriva dessa kontroller på. Antingen på ett sätta där man döper funktionerna på ett speciellt vis, exempelvis *action_index*, för att sedan i routingen styra vad som ska hända beroende på vilken typ av HTTP-request det är. Att annat sätt att skriva dessa namn på, är att i själva funktionsnamnet bestämma om den exempelivs tillhör en GET-request. Dessa skriver man exempelvis som *get_index*, och skriver en speciell bit kod längst upp i kontrollerklassen, `public $restful = true;`. Då behöver man i sin routingfil endast registrera sin kontroller, exempelvis `Route::controller('posts');`, för att registrera sin kontroller med namnet *Posts_Controller*.

Laravel kommer med ett inbyggt autentiseringssystem, som kräver väldigt lite för att få det att fungera. Det som krävs för att använda sig av detta system, är en tabell för användare. Denna tabell ska som minst innehålla ett fält för id, användarnamn/epost och lösenord. Som standard förväntar sig Laravel att denna tabell ska heta *users*, med modellen *User* och fälten *id, email* och *password*. Dessa namn går det enkelt att ändra till sina egna, i filen *auth.php* som ligger i mappen *application/config*.

För att kontrollera om en användare är inloggade eller inte, och därmed begränsa tillgången till vissa routes, så finns det ett filter i *routes.php* som heter *auth*. Genom att i sin route då skriva att detta filter ska köras innan routen exekveras, så kan routes begränsas beroende på om besökaren är inloggad eller inte.

Exempel på route som kräver inloggning:

```
Route::get('admin', array('as' => 'new_post', 'before' => 'auth', 'uses' =>
'admin@new'));
```

View

Vy-lagret består av de sidor som kommer visas, med olika innehåll beroende på vad som sker i routingen/kontrollerna. De vyfiler tillhörande applikationen har en designerad mapp, *views*, som både modellerna samt kontrllerna har. Här i kan man sedan själv dela upp sina filer i olika undermappar, beroende på till vilken del av applikationen filerna hör. I denna exempelapplikation, så har jag skapat en mapp med namnet *admin* och en mapp med namnet *post*. I mappen *admin* kommer vyn innehållandes formuläret för att logga in ligga, samt vyn innehållandes formuläret för att skapa nya inlägg.

Laravel har en vy-motor kallad Blade, som bland annat gör att applikationens vyer kan delas upp i mindre delar. Flera sådana mindre filer kan sedan användas i samma dokument, och senare återanvändas av andra dokument. Det finns även möjlighet att använda sig utav template-filer, för att kunna få exempelvis samma grundstruktur på en rad olika sidor, utan att behöva skriva samma bit kod i alla dessa filer. Vy-filer som ska nyttja blade-motorn döps med filändelsen *blade.php*.

För att skriva ut något i en Blade-fil, så används *{{}}*, vilket blir motsvarigheten till att skriva *<?php ?>*. Det går även att enkelt använda *if*-satser och *for*-loopar i dessa filer, för att skriva ut den data som man hämtat i sin kontroller. Det finns även en rad olika hjälpfunktioner för att skapa HTML-element, så som till exempel länkar, bilder, formulär och fomrulärkontroller. För att exempelvis skapa en länk till en route, kan man använda *HTML::link_to_route*, där man förutom att skicka in namnet till den route man vill att länken ska gå till och vad som ska stå på länken, även kan skicka in parametrar som ska finnas i adressen, och en array med HTML-attribut, så som class och id.

Exempel på hur alla poster kan skrivas ut:

```
@forelse ($posts as $post)
	<div class="post">
		<h2>
			{{ HTML::link_to_route('view_post', $post->title, $post->id) }}
		</h2>
		<p>
			{{ substr($post->content, 0, 120).' [...]' }}
		</p>
		<p>
			{{ HTML::link_to_route('view_post', 'Read more &rarr;', $post->id) }}
		</p>
	</div>
@empty
	<h2>
		No posts
	</h2>
@endforelse
```

Vyer kan skapas både direkt från en route, och från en funktion i en kontroller. I denna exempelapplikation har jag valt att skapa vyn från kontrollern, för att följa MVC-mönstret.

Exempel på att skapa en vy:

```
return view::make('post.index', $context);
```

I exemplet ovan skapas vyn *index.blade.php*, som ligger i mappen *views/post*. Den andra paramtern som skickase med, är en array med alla de data som kommer användas i vyn.

Slutsats

Efter att genom denna rapport ha testat på ramverkete Laravel, så kan jag säga att jag definitivt vill arbeta mer med det, då jag fann det väldigt enkelt, men samtidigt väldigt kompetent, att arbeta med. Detta gäller allt från routingsystemet till vy-motorn Blade, som under lättar väldigt mycket när det kommer till att skriva ut data till användaren. Jag har själv inte sett något sätt som jag skulle kunna påstå är bättre och mer lättanvänt än detta.

Angående kontrollerna, så har jag ännu inte testat på att använda mig av vad Laravel kallar *restful controllers*, där man istället för att döpa kontrollerfunktionerna med *action_* i början, lägger till en HTTP-requesttyp, exempelvis *get_* eller *_post*. Jag fann det väldigt enkelt och bra att arbete med typen av anrop direkt i routes-filen istället.

Jag fann även att ramverkets modellhanterare, Eloquent, var väldigt enkel att arbeta med, åtminstone i denna lilla exempelapplikation. Hur väl det fungerar senare, i större applikationer med en mer komplex databas och relationer, återstår att se. Men jag tror att det även då kommer kunna vara ett väldigt bra hjälpmedel. Även sättet man skapara databasens tabeller på, med hjälp av migrations, tyckte jag var väldigt smidigt.

Vy-motorn, Blade, var som jag tidigare skrev, säkerligen en av de funktionerna jag tyckte allra mest om att arbeta med. Mycket på grund av sin enkelhet, med att den ändå är väldigt kompetent och kan snabba på arbetet med att skriva ut information från modellerna man hämtat i sin kontroller. Det tillsammans med möjligheten att dela upp det i flera mindre filer, exempelvis någon fil för menyn, någon fil för en footer, och sedan ha en slags layout-fil, för att enkelt kunna få samma upplägg på alla sidor. Även alla de metoder för att skapa HTML-element som finns, exempelvis *HTML::link_to_rout*, fanna jag väldigt användbara. Om man tar *link_to_route* som exempel, så använder den namnet på routen, inte själva adressen. Så om man då senare bestämmer sig för att ändra adressen på en namngiven route, så behöver man endast ändra i *routes.php*, och inte gå in i varje fil där man använts sig av den adressen.

En annan del som jag inte tog upp i denna rapport, som är en viktig del för alla typer av programvaror och ramverk, är dokumentationen. Även här är så klarar sig Laravel bra, då jag tyckte att alla delar av Laravel var väldokumenterade, dels i form av den dokumentation som finns på deras sida, och dels i form av kommenterare i koden. Exemplevis så följer det med kommenterarer i såväl *routes.php* som i den exempelkontroller som följer med när man laddar ner ramverket.

Sammanfattningsvis så tyckte jag att Laravel verkade vara ett väldigt enkelt och kompetent ramverk,

som jag definitivt kommer kolla mer på och testa att bygga större, mer komplexa, applikationer.

Bilagor

Vyer

views/template.blade.php

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      @yield('title')
    </title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    {{ Asset::styles() }}
    {{ Asset::scripts() }}
  </head>
  <body>
    <div class="container" id="main_container">
      @if (Auth::user())
        {{ HTML::link_to_route('logout', 'Logout') }}
      @else
        {{ HTML::link_to_route('login', 'Login') }}
      @endif
      @yield('main_content')
    </div>
  </body>
</html>
```

views/post/index.blade.php

```
@layout('template')
@section('title')
      Blogg
@endsection
@section('main_content')
      @forelse ($posts as $post)
            <div class="post">
                  <h2>
                        {{ HTML::link_to_route('view_post', $post->title,
$post->id) }}
                  </h2>
                  <p>
                        {{ substr($post->content, 0, 120).' [...]' }}
```

```
                </p>
                <p>
                        {{ HTML::link_to_route('view_post', 'Read more
&rarr;', $post->id) }}
                </p>
            </div>
        @empty
            <h2>
                No posts
            </h2>
        @endforelse
@endsection
```

views/post/post.blade.php

```
@layout('template')
@section('title')
    Single post
@endsection
@section('main_content')
    <div class="post">
            <h1>
                {{ HTML::link_to_route('view_post', $post->title,
$post->id) }}
            </h1>
            <p>
                {{ $post->content }}
            </p>
            <p>
                {{ $author->username }}
            </p>
            <p>
                {{ HTML::link_to_route('home', '&larr; Back to index') }}
            </p>
        </div>
@endsection
```

views/admin/login.blade.php

```
@layout('template')
@section('title')
    Login
@endsection
@section('main_content')
    {{ Form::open(URL::to_route('login_post'), 'POST', array('id' =>
'login_form_wrapper')) }}
        {{ Form::token() }}
        @if (Session::has('login_errors'))
            <span class="error">Username or password incorrect.</span>
        @endif
        <div class="field">
            <p>{{ Form::text('username', '', array('class' =>
'input-block-level', 'placeholder' => 'username')) }}</p>
        </div>
        <div class="field">
            <p>{{ Form::password('password') }}</p>
        </div>
        <div class="action">
            <p>{{ Form::submit('Login', array('class' => 'btn btn-large
btn-block btn-primary')) }}</p>
        </div>
    {{ Form::close() }}
@endsection
```

views/admin/new.blade.php

```
@layout('template')
@section('title')
    New post
@endsection
@section('main_content')
    <h1>New post</h1>
    <h2>Welcome, {{ Auth::user()->username }}!</h2>
    {{ Form::open('admin') }}
      {{ Form::hidden('author_id', $user->id) }}
        <p>{{ Form::label('title', 'Title') }}</p>
        {{ $errors->first('title', '<p class="error">:message</p>') }}
        <p>{{ Form::text('title', Input::old('title')) }}</p>
        <p>{{ Form::label('content', 'Content') }}</p>
        {{ $errors->first('content', '<p class="error">:message</p>') }}
        <p>{{ Form::textarea('content', Input::old('content')) }}</p>
        <p>{{ Form::submit('Create', array('class' => 'btn btn-primary'))
}}</p>
    {{ Form::close() }}
@endsection
```

Kontroller

controllers/post.php

```
<?php
class Post_Controller extends Base_Controller {

	public function action_index()
	{
		$posts = Post::with('author')->order_by('created_at',
'desc')->get();
		$context = array('posts' => $posts);
		return view::make('post.index', $context);
	}

	public function action_new()
	{
		return view::make('post.new');
	}

	public function action_view($post_id)
	{
		$post = Post::find($post_id);
		$author = $post->author;
		$context = array('post' => $post, 'author' => $author );
		return view::make('post.post', $context);
	}
}
?>
```

controllers/admin.php

```
<?php
class Admin_Controller extends Base_Controller {

	public function action_index()
	{
		return view::make('admin.login');
	}

	public function action_login()
	{
		$userdata = array(
			'username' => Input::get('username'),
			'password' => Input::get('password')
		);

		if (Auth::attempt($userdata)) {
			return Redirect::to_route('new_post');
		}

		else {
			return Redirect::to_route('login')
				->with('login_errors', true);
		}
	}

	public function action_logout()
	{
		Auth::logout();
		return Redirect::to_route('home');
	}

	public function action_new()
	{
		$user = Auth::user();
		$context = array('user' => $user);
		return view::make('admin.new', $context);
	}

	public function action_create()
	{
		$new_post = array(
			'title' => Input::get('title'),
			'content' => Input::get('content'),
			'author_id' => Input::get('author_id')
		);

		$rules = array(
			'title' => 'required|min:3|max:128',
			'content' => 'required'
```

```
            );

            $v = Validator::make($new_post, $rules);

            if ($v->fails()) {
                  return Redirect::to_route('new_post')
                        ->with('user', Auth::user())
                        ->with_errors($v)
                        ->with_input();
            }

            $post = new Post($new_post);
            $post->save();
            return Redirect::to_route('view_post', $post->id);
      }
}
?>
```

Modeller

models/post.php

```
<?php
class Post extends Eloquent
{
	public function author()
	{
		return $this->belongs_to('User', 'author_id');
	}
}
?>
```

models/user.php

```
<?php
class User extends Eloquent
{
	public function posts()
	{
		return $this->has_many('Post');
	}
}
?>
```

Migrationer

migrations/2013_03_30_132740_create_users.php

```php
<?php
class Create_Users {
	/**
	 * Make changes to the database.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('users', function ($table) {
			$table->increments('id');
			$table->string('username', 128);
			$table->string('nickname', 128);
			$table->string('password', 64);
			$table->timestamps();
		});

		DB::table('users')->insert(array(
		    'username'  => 'admin',
		    'nickname'  => 'Admin',
		    'password'  => Hash::make('admin')
		));
	}

	/**
	 * Revert the changes to the database.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('users');
	}
}
```

migrations/2013_03_30_132850_create_posts.php

```php
<?php
class Create_Posts {
	/**
	 * Make changes to the database.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('posts', function ($table) {
			$table->increments('id');
			$table->string('title', 128);
			$table->text('content');
			$table->integer('author_id');
			$table->timestamps();
		});
	}

	/**
	 * Revert the changes to the database.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('posts');
	}
}
```