Building out RESTful Controller methods in Laravel 4

POSTED BY PHILIP BROWN ON AUGUST 12TH, 2013
0 COMMENTS

Over the past couple of weeks, I've looked at setting up a Controller, making your Controllers flexible, and testable and we've looked at using Mockery in order to write tests.

The time has finally come to actually start building out the methods of a RESTful Controller.

In this post I'm going to go through building a RESTful Controller step by step.

Setting the stage

So if you remember back to the previous tutorials, I'm using a User Repository and injecting it through the Constructor. So far the `UsersController.php` file should look like this:

```
<?php

use Cribbb\Storage\User\UserRepository as User;

class UsersController extends BaseController {

  /**
   * User Repository
   */
  protected $user;

  /**
   * Inject the User Repository
   */
  public function __construct(User $user)
  {
    $this->user = $user;
  }
```

I've also already created the `index()` method:

```
/**
 * Display a listing of the resource.
 *
 * @return Response
 */
public function index()
{
  return $this->user->all();
}
```

Currently, this simply returns JSON. I'm not too worried about creating a view and displaying this data property for the time being, so this method is effectively finished so far.

Create

So the first method I'm going to look at is `create();`. The Create method is

used for displaying a form to create a new resource. This is a `GET` request and so we are simply getting a View, we don't want to actually create a new user using this method.

Creating a new user should probably be tied with registration, but we will deal with that at some point in the future. For now, just think of a User being a typical resource.

So the only thing to do in the `create();` method is to specify which View should be rendered by Laravel:

```
/**
 * Show the form for creating a new resource.
 *
 * @return Response
 */
public function create()
{
  return View::make('users.create');
}
```

Remember when we created a new resource in `routes.php`?

```
Route::resource('users', 'UsersController');
```

This means Laravel will automatically be expecting `/users/create` as a route so we don't have to define it.

Now fire up the server and hit the `/users/create` URL in your browser:

```
$ php artisan serve
```

You should be greeted with an error, `View [users.create] not found.`.

In order to display a View, first we have to actually create it. Create a new directory under `app/views` called `users`. Next create a new file called

`create.blade.php` and type some stuff so you will know that it is working when you see it in the browser.

When I wrote:

```
1 return View::make('users.create');
```

This means Laravel will be looking for a file called `create` under the `users` directory. You can think of `users.create` as simply `users/create`. Also, I have given the `create` View a blade extension. This is because I'm going to be using Laravel's templeting engine. I haven't really covered blade yet, but I will in a future tutorial. Blade simply allows you to make better HTML templates for your views.

So now that you have created `app/views/users/create.blade.php` reload the page in your browser and everything should work.

To test this, we can simply create a new test like this:

```
1 public function testCreate()
2 {
3   $this->call('GET', 'users/create');
4
5   $this->assertResponseOk();
6 }
```

Here I'm just making a request to the page and asserting that the response is 200.

Store

The next method to look at is `store();`. To create a new resource, we POST to the store method.

The store method of the Controller should only be concerned with accepting the data from the POST request. Remember, Controllers should only be

concerned with the flow of traffic.

If you are using my Magniloquent package, all your validation should be abstracted away into the model. This means we can simply test to see if the data is saved and then act accordingly:

```
/**
 * Store a newly created resource in storage.
 *
 * @return Response
 */
public function store()
{
  $s = $this->user->create(Input::all());

  if($s->isSaved())
  {
    return Redirect::route('users.index')
      ->with('flash', 'The new user has been created');
  }

  return Redirect::route('users.create')
    ->withInput()
    ->withErrors($s->errors());
}
```

In this method I'm simply attempting to store the data. If the record passes the validation rules and is saved correctly, we can redirect back to the Controller's index method and pass a flash message as a confirmation.

```
if($s->isSaved())
{
  return Redirect::route('users.index')
    ->with('flash', 'The new user has been created');
}
```

If the record fails to save, we can just redirect back to the create method and send the input and the errors so that the user can fix their mistake and try again.

To test this method, we only need to assert that the user is redirected to the correct place depending on success or failure. Remember, we should only be testing one thing at a time and so we need to mock the other dependancies:

```
1  public function testStoreFails()
2  {
3    $this->mock->shouldReceive('create')
4               ->once()
5               ->andReturn(Mockery::mock(array(
6                   'isSaved' => false,
7                   'errors' => array()
8                 )));
9
10   $this->call('POST', 'users');
11
12   $this->assertRedirectedToRoute('users.create');
13   $this->assertSessionHasErrors();
14 }
```

In this test I'm mocking that the save failed and that the user was redirected back to the create method with errors.

```
1  public function testStoreSuccess()
2  {
3    $this->mock->shouldReceive('create')
4               ->once()
5               ->andReturn(Mockery::mock(array(
6                   'isSaved' => true
7                 )));
8
9    $this->call('POST', 'users');
10   $this->assertRedirectedToRoute('users.index');
11   $this->assertSessionHas('flash');
12 }
```

And in this method I'm asserting that the user is redirected to the index method with a flash message.

Show

The show method simply returns a single record based on an id that is

passed through the URL.

```
/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return Response
 */
public function show($id)
{
  return $this->user->find($id);
}
```

And this is the test to ensure things are working correctly:

```
public function testShow()
{
  $this->mock->shouldReceive('find')
             ->once()
             ->with(1);

  $this->call('GET', 'users/1');

  $this->assertResponseOk();
}
```

Edit

The edit method needs to return a view that contains the form to allow a user to edit the resource. Much like the create method, all we need to do is return the view.

First create a new view under `app/views` called `edit.blade.php`.

Next, update the edit method to return the view:

```
/**
 * Show the form for creating a new resource.
 *
 * @return Response
 */
```

```
6  public function edit()
7  {
8    return View::make('users.edit');
9  }
```

And finally, the test to ensure the route responses correctly:

```
1  public function testEdit()
2  {
3    $this->call('GET', 'users/1/edit');
4
5    $this->assertResponseOk();
6  }
```

Update

To update a user, all we have to do is find the user by the id and then attempt to update the model. Again much like the store method, the Controller is only really concerned with directing traffic:

```
 1  /**
 2   * Update the specified resource in storage.
 3   *
 4   * @param  int  $id
 5   * @return Response
 6   */
 7  public function update($id)
 8  {
 9    $s = $this->user->update($id);
10
11    if($s->isSaved())
12    {
13      return Redirect::route('users.show', $id)
14        ->with('flash', 'The user was updated');
15    }
16
17    return Redirect::route('users.edit', $id)
18      ->withInput()
19      ->withErrors($s->errors());
20  }
```

I added the following method to the `UserRepository`:

```
public function update($input);
```

And the following method to `EloquentUserRepository`:

```
public function update($id)
{
  $user = $this->find($id);

  $user->save(\Input::all());

  return $user;
}
```

And finally, the two tests are as follows:

```
public function testUpdateFails()
{
  $this->mock->shouldReceive('update')
             ->once()
             ->with(1)
             ->andReturn(Mockery::mock(array(
                 'isSaved' => false,
                 'errors' => array()
               )));

  $this->call('PUT', 'users/1');

  $this->assertRedirectedToRoute('users.edit', 1);
  $this->assertSessionHasErrors();
}

public function testUpdateSuccess()
{
  $this->mock->shouldReceive('update')
             ->once()
             ->with(1)
             ->andReturn(Mockery::mock(array(
                 'isSaved' => true
               )));

  $this->call('PUT', 'users/1');

  $this->assertRedirectedToRoute('users.show', 1);
  $this->assertSessionHas('flash');
```

```
30 }
```

Delete

Finally, the delete method simply searches for a record based on the id and deletes it:

```
1  /**
2   * Remove the specified resource from storage.
3   *
4   * @param  int  $id
5   * @return Response
6   */
7  public function destroy($id)
8  {
9    return $this->user->delete($id);
10 }
```

In my `EloquentUserRepository.php` file, the delete method is simply:

```
1 public function delete($id)
2 {
3   $user = $this->find($id);
4
5   return $user->delete();
6 }
```

Conclusion

And there you have it, your first basic RESTful controller. As you can see, by following the conventions of REST, we can very quickly build Controllers that meet all the basic requirements of a simple web application.

Whilst this is most certainly a typical example of a RESTful controller, it should provide you with a good basis for building your own RESTful controllers in your projects. You will likely come up with some strange scenarios that you need to solve, but by following the conventions of REST, you will have a solid foundation to begin with.

This is a series of posts on building an entire Open Source application called Cribbb. All of the tutorials will be free to web, and all of the code is available on GitHub.

So far I've covered:

1. Getting started with Laravel 4
2. Laravel 4 Migrations
3. Setting up your first Laravel 4 Model
4. Getting started with testing Laravel 4 Models
5. Laravel 4 Fixture Replacement with FactoryMuff
6. Creating the Twitter following model in Laravel 4
7. Laravel 4 Eloquent Model Relationships
8. Setting up Vagrant with Laravel 4
9. Creating a Laravel 4 package
10. Setting up your first Laravel 4 Controller
11. Creating flexible Controllers in Laravel 4 using Repositories
12. How to structure testable Controllers in Laravel 4
13. Getting started with Mockery
14. Creating Laravel 4 Validation Services
15. Extending Eloquent in Laravel 4

The next post in this series is Creating forms in Laravel 4

Philip Brown

Hey, I'm Philip Brown, a designer and developer from Durham, England. I create websites and web based applications from the ground up. In 2011 I founded a company called Yellow Flag. If you want to find out more about me, you can follow me on Twitter or Google Plus.

Join the Culttt

Become an insider and join the **Culttt**

Your email...

Join us

hi Philip - just a quick one please.

public function testCreate()

{
$this->call('GET', 'users/create');
$this->assertResponseOk();
}

I'm finding if I put anything in the call() function it passes the test...
eg. call('GET', 'users/grejsguirhsg') comes back with no errors.
similarly if I put this in to my url it comes back with a blank page and no reported error.

Have you seen this before?
I'm doing something daft, aren't I?

Thanks.
DS

• Reply • Share ›

dstewart101 ➚ dstewart101 • a month ago

Ok ... seemingly all works as expected until I make a blade template- I must have fouled up somewhere. I'll work with this again, and see what I come up with.

• Reply • Share ›

Philip Brown Mod ➚ dstewart101 • a month ago

Do you have a catch-all route? Or is there any error in your logs? Yeah, it will just be something really small that is causing the weirdness :)

• Reply • Share ›

David Stewart ➚ Philip Brown • a month ago

erg. feel like a wally now. typo in the "use app\Storage\User\UserRepository as User;" line of my controller. thanks for encouraging me to look harder.
thanks again for the follow up.

• Reply • Share ›

Philip Brown Mod ➚ David Stewart • a month ago

Haha, don't worry about it, it happens every day to

me.

• Reply • Share ›

Snapey • 4 months ago

I've checked all my folders and can't find Laravels 'tempting engine' :-0

• Reply • Share ›

Philip Brown Mod ➔ Snapey • 4 months ago

Haha, you sure? It's pretty tempting :p

1 • Reply • Share ›

vineet ➔ Snapey • 6 days ago

Pretty Templating! :p

• Reply • Share ›

Philip Brown Mod ➔ vineet • 6 days ago

haha, I'll never live that down :P

• Reply • Share ›

mat • 4 months ago

Hey boss - just a small thing I spotted. The update tests seem to be returning a mock with 'passes', and believe you mean to use 'isSaved'

• Reply • Share ›

Philip Brown Mod ➔ mat • 4 months ago

Haha, oops, yeah you're right. That was the name of the old method.

I should employ you to catch my mistakes :p haha.

Thank you again sir :)

• Reply • Share ›

Join the Culttt

Become an insider and join the **Culttt**

Your email...

Join us

Become a follower

- Twitter
- Facebook
- Google Plus
- RSS