(/)

April 21, 2015

Hello world, I am Laravel (5)

So there is this thing called Laravel. You may have heard of it already, but you're not sure what it is actually about? Or you do, but want to know more about it and its great new features in version 5? Great, this post is especially for you! Laravel is at the same time one of the youngest and most popular PHP frameworks out there. So how does this work together? Let us take a closer look at why it is that popular and how it could be of use for you too. We will go through the main functionalities and talk about brand new features in version 5.

Intro

Laravel is a PHP full-framework for all kinds of applications. It is built on modern PHP standards and best practices with the goal to make high-level coding fun. It also tries to make starting a new application as easy as possible while keeping flexibility.

For the second time in a row Laravel has won the SitePoint PHP framework survey (http://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results). Let us take closer look at this framework's components and why you should absolutely know about it.

Routes

A routes component is fundamental to every framework. It takes the requested URL and decides where to go next.

In the first example we specify that we want to return a "Hello World" string when the user requests the applications root. (e.g. www.myapplication.com).

```
Route::get('/', function()
{
    return 'Hello World';
});
```

But there is more. Now we are calling a controller method for a URL like "myapp.com/users/1". The "{id}" is a parameter which will be available in the method. In this example the "{id}" would be "1".

```
Route::get('user/{id}', 'UserController@showProfile');
```

In both examples we are listening for a GET request. Of course you can use all the other request methods (POST DELETE or PUT) like this too.

```
// More route examples
Route::post('user', 'UserController@store');

Route::put('user/{id}', 'UserController@udpate');

Route::delete('user/{id}', 'UserController@delete');
```

Now that we know about the roads we can go, let's grab some data on the way.

Eloquent

Eloquent is the Object Relational Mapper of Laravel(ORM). ORMs allow us to map our application objects to database tables. For example if we are building a blog, then posts, photos pages or comments would be such objects. The ORM will take care of handling these objects so we do not have to write any line of SQL.(unless we want to)

The only thing we need for working with Eloquent is a model per database table that extends the Model class.

```
class User extends Model {

    protected $table = 'my_users';

}
```

If the model class is named "User", Laravel will automatically look for a "users" table in the database. To overwrite this behavior we can add $table with the name of the table.

Now we are good to go for some basic Eloquent magic.

```
// Get all users
User::all();

// Get a specific user by id
User::find(1);

// I guess you know by now that you do not need these comments
User::destroy(1);

User::where('name', 'Jim')->first();
```

Aren't these calls really elegant and nice to read? You immediately know what we are doing while reading the code. Let's do some more for creating, reading, updating and deleting. (so called basic CRUD functions)

```
// Create
$user = new User;
$user->name = 'Jim';
$user->save();

// Read
$user = User::where('name', 'Jim')->first();

// Update
$user->update(['name' => 'John']);

// Delete
$user->delete();
```

This is only the tip of the iceberg. There is so much more you can do with Eloquent like working with relations, eager loading and collections.

> **Note:** Using Laravel's ORM is not a must, but I haven't had a project without it.

Facades

In Laravel you have access to framework objects through Facades. They look like calling static methods, but that's not completely the case. In fact a non-static method is called under the hood. Laravel just makes use of the simple syntax. This can be a little bit confusing at the beginning but it also makes using these classes damn simple.

```
// Using Laravel Cache Facade
$value = Cache::get('key');

// Using Laravel Auth Facade
if (Auth::check())
{
    // The user is logged in...
}
```

You can also define Facades yourself. A lot of Laravel packages are using them too. So remember, Facades are little helpers for calling Laravel classes.

Artisan

Every good framework has a command-line interface(CLI). Laravel's is called Artisan and is build on the Symfony console. Artisan provides helpful commands for your developing process. There is a lot it can do for you and here are some examples.

```
// List all Artisan command
php artisan list

// Cache your routes
php artisan route:cache

// Or create a migration (there is a sperate section about migrations)
php artisan migrate:make create_users_table
```

Make sure to checkout all the other Artisan commands too. They will help you develop faster and you do have to take care of the boring stuff.

Controller

As we have already seen it is possible to respond to routes directly in our `routes.php` file. This is for some cases ok, but we want a cleaner way. Controllers will help us structuring our applications. Our routes will trigger controller methods to perform certain jobs before we return data or a whole view.

The controller example receives an `$id` from our route `Route::get ('user/{id}', 'UserController@showProfile');` from before. Then it returns a view with the data of the user with the given id.

```
<?php namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('profile', ['user' => User::findOrFail($id)]);
    }

}
```

Resource Controller

There is a good chance that when we are working with our database objects (resources in this case), that we want to perform almost the same actions again and again. Let's assume we have photos in our app. Think about what you want to do with them:

- show all photos
- show a form to create a photo
- store a photo
- show a specific photo
- show a form to edit a specific photo
- update a specific photo
- delete a specific photo
- ...

This is true for photos, posts, comments and other typical resources. In a lame world we would have to write routes for all theses actions that are linked to controller methods. This would look something like that.

```
Route::get('/photo', 'PhotoController@index');

Route::get('/photo/create', 'PhotoController@create');

Route::post('/photo', 'PhotoController@store');

Route::get('/photo/{id}', 'PhotoController@show');

Route::get('/photo/update/{id}', 'PhotoController@edit');

Route::delete('/photo/{id}', 'PhotoController@destroy');

...
```

I think you got it. It's a lot to write and to repeat. Now think about the controller and its methods you would have to write too.

This is where Laravel's `RESTful resource controller` comes handy. An Artisan command will trigger the magic.

```
php artisan make:controller PhotoController
```

This will do two things. First it will create a PhotoController with methods for index, create, store, show, edit, update and destroy. And secondly you can use the resource route for connecting all these methods to routes.

```
// Defining a resource route
Route::resource('photo', 'PhotoController');
```

The table below shows the created routes and the connected controller methods . This will save you a lot of time and is another great Laravel feature that helps you concentrate on building your app.

Verb	Path	Action	Route Name
GET	/photo	index	photo.index
GET	/photo/create	create	photo.create
POST	/photo	store	photo.store
GET	/photo/{photo}	show	photo.show
GET	/photo/{photo}/edit	edit	photo.edit
PUT/PATCH	/photo/{photo}	update	photo.update
DELETE	/photo/{photo}	destroy	photo.destroy

Migrations

We all have been there. You are starting a project and after some time you notice that you need another table or column. Ok, you could just add it to your local database. So where's the big deal?

> ***"Migrations are like version control for your database structure. "***

Another developer gets your latest code from version control, but `not` the changes to the database structure. Not until migrations.

They are just files describing your changes to to the database structure. Simply like "I added a users table with the columns..." or "I changed the description column from the recipes table from a varchar field to a text field". These files live inside you repository and therefor you can push them with your code so that other developers know about them too.

```
<?php

// Imports removed for example
class CreateUsersTable extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up() {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('email')->unique();
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down() {
        Schema::drop('users');
    }

}
```

This is what a typical migration file looks like. It's a class with an up and a down method. In the first one we are defining the changes we would like to make to our database. (like adding, removing or changes tables or columns)

In the down method we are undoing the changes from the up method. This is because we would like to be able to roll changes back in some cases. In our case deleting the new table would be the opposite of creating it.

In Laravel you can run migrations with the artisan command `php artisan migrate`. It will check for new migrations and will change your database structure if needed. I wouldn't want to work without them anymore!

Form requests

This is my favorite Laravel 5 feature and yeah, it has something to do with forms. It's a clean way to validate your forms through custom classes.

To create a new form request use the artisan command `php artisan make:request StorePhotoRequest`. The name implies that we are using this request to validate a new photo's data. Let's take a look at the two important methods of the created class.

```
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
     public function authorize() {
         return \Auth::check();
     }

     /**
      * Get the validation rules that apply to the request.
      *
      * @return array
      */
     public function rules() {
         return [
             'title' => 'required|string',
             'description' => 'required|string',
         ];
     }
```

In the authorize method we determine if the request is authorized. (surprise!) False will return a 403 HTTP response, true will let the request pass. In our authorize method we check if the user is logged in.

The rules method returns an array with the typical Laravel validation rules. We are defining some columns and their requirements like the title is a must und needs to be a string.

Ok got it, but why is this so special you might think? First a separate class is the best way to handle these validations, but there is more. Let's see how we trigger the validation.

```
// From the PhotoController
  public function store(StorePhotoRequest $request)
  {
      // The incoming request is valid...
  }
```

Yep, again type hinting the validation class is enough to trigger the validation before the request hits the method. If the validation fails, the request will never get to the inside of the method. Pretty amazing right?

Note: Form requests are a brand new Laravel 5 feature

Dependency Injection

Dependency Injection is a design pattern that avoids hard-coding dependencies. It helps making your application more modular and prevents you from having tightly coupled classes. This is helpful for testing too, because you can easily mock the dependency.

> ***"Dependency Injection is a key element of agile architecture - Ward Cunningham"***

```
class UsersController extends BaseController
{
    public function getIndex()
    {
        return Response::make('Hello users!’);
    }
}
```

In this user controller wer are using the Response class (yeah right, it's a Facade too) inside the getIndex method. This is totally ok when you're building a small project and you do not need / want to test this class. But if not, this is bad practice. Our UserController class is `tightly coupled` to the Response class. This is what we want to avoid through dependency injection.

Constructor injection

One way to solve that would be with constructor injection.

```
class UsersController extends BaseController
{
    protected $response;

    public function __construct(Response $response)
    {
        $this->response = $response;
    }

    public function getIndex()
    {
        return $this->response->make('Hello Users!’);
    }

}
```

Now we inject the Response class within the constructor through type hinting the class. Laravel knows that you need an object of that class and will get that for you. Therefor our UserController is now loosely coupled to the Response class. We can easily switch it out and mock it for tests. Perfect!

Note: Type hinting is when you force parameters to be a specific type.

Method injection

Probably you do not need the Response class in all your UserController methods. In this case method injection comes handy. We can inject the class within our getIndex method arguments. Now we do not need the constructor anymore and we save some lines.

```
class UsersController extends BaseController
{

    public function getIndex(Response $response)
    {
        return $response->make('Hello World!’);
    }

}
```

Note: Method injection is a brand new Laravel 5 feature

Conclusion

I hope I could show how great it is working with Laravel on your applications. Everything is built to help you working on your projects and to make working fast and fun. Of course these were just the base components you need to know about as a start. It is getting even better when you find all the little tricks and features that are available for you. So go and create something amazing now!

More Laravel stuff

Elixir

Elixir is an API for writing Gulp tasks in Laravel. Gulp is a JavaScript task manager like Grunt. You can manage your assets and automate theses tasks. (minify, concatenate, optimise images, etc.) Read more (http://laravel.com/docs/5.0/elixir)

Homestead

Homestead is a ready to go vagrant box especially optimized for Laravel. Read more (http://laravel.com/docs/5.0/homestead)

Laracasts

It's the best PHP screencasts on the web. That's what the slogan sounds like, and it really is! There is no better web dev teacher I know than Jeffrey Way . He totally understands what it is like learning new stuff and he is always up to date with Laravel, PHP and other imported web stuff. I can totally recommend signing Laracasts. Read more (https://laracasts.com)

Forge

Forge is a hosting platform for your Laravel sites. It's like a remote Homestead box with a lot of features for your deployment. Read more (https://forge.laravel.com)

Envoyer

Envoyer is a zero downtime PHP deployment service. It focuses on a simple deployment process for your applications and is perfect in combination with Forge. Both Forge and Envoyer are not Laravel specific. Read more (https://envoyer.io)

Lumen

Lumen is the brand new Laravel micro-framework. It is built to be extremely fast and is faster than other known light frameworks like Slim oder Silex. Every time you need a microservice you can use Lumen without sacrificing the power of Laravel. You can still use your favorite ORM eloquent and more, but without the overhead of a full-framework. If you at some time need your application to do more, you can easily switch Lumen out with the standard Laravel. That's awesome and something great for the community. Read more (http://lumen.laravel.com)

6 Comments Christoph Rumpel - Passionate Web Developer 1 Login

Recommend 1 Share Sort by Best

Join the discussion…

freddy • 8 hours ago
Cool i just new bie at this case, i'm coming with codeignter :D
• Reply • Share ›

Christoph Rumpel Mod freddy • 7 hours ago
Ah I've been there before. I also started with Codeigniter before they stopped developing it. Glad you liked the article! Thx.
• Reply • Share ›

freddy Christoph Rumpel • 7 hours ago
Lol seems like you mock at Codeignter , they need some refresh might be, honestly i'm still use codeignter for old project :p

Freddy Sidauruk
• Reply • Share ›

Christoph Rumpel Mod freddy • 3 hours ago
Everybody should use what fits the needs :-) But CI version 3 was released last month. Let's see where this is going.
• Reply • Share ›

Luciano Mammino • 5 days ago
Great article. It's nice to have this resume of all the Laravel capabilities and products related to its ecosystem. I'm sure it will help a lot of newcomers to understand what is possible with this frameworks and especially people like me who are coming from other

frameworks (Symfony, Silex).

PS: It's scheduled on my hootsuite □

• Reply • Share ›

Christoph Rumpel Mod Luciano Mammino • 5 days ago

Thx! Yeah that's exactly how I see it too :-)

1 • Reply • Share ›

ALSO ON **CHRISTOPH RUMPEL - PASSIONATE WEB DEVELOPER** WHAT'S THIS?

WebP: One image format to rule them all

4 comments • 2 years ago

Avat **Christoph Rumpel** — Thanks a lot! Yeah that's what i thought too. It is time for a new format, but until it gains acceptance

How to Laravel series: What is new in Laravel 4.1

2 comments • a year ago

Avat **Christoph Rumpel** — Yeah, this is really a great new feature!

Redesigning christoph-rumpel.com

2 comments • 2 years ago

Avat **Christoph Rumpel** — Thanks Hugo!

Front-end performance part 01: Assets loading

5 comments • 2 years ago

Avat **thomas piribauer** — comprehensive overview, covered the topic very well. nice read!

Built with Jekyll (http://jekyllrb.com/), Bootstrap (http://getbootstrap.com/) and Love (http://en.wikipedia.org/wiki/Love)
Find me on Twitter (http://twitter.com/christophrumpel), GitHub (https://github.com/christophrumpel), Treehouse (http://teamtreehouse.com/christophrumpel), Google (https://plus.google.com/u/0/116404537718858902366/posts) or RSS (http://christoph-rumpel.com/feed.xml)