Create your own MVC framework in PHP

Noufel Gouirhate Follow
Dec 17, 2017 · 5 min read ★

Before learning MVC, I was developing websites in a linear way. Naturally, I created a php file for each page. And each file was a mix of php and html… really nasty mix. The more we were moving into the project and the more we had difficulties in maintaining it: a lot of redundancy in the HTML structure, difficulty to read the code because of the php directly inserted in the view… Even if the project ended in good condition, we really started to get mixed up with our logic.

> *You can follow this tutorial with my repo Github : https://github.com /ngrt/MVC_todo*

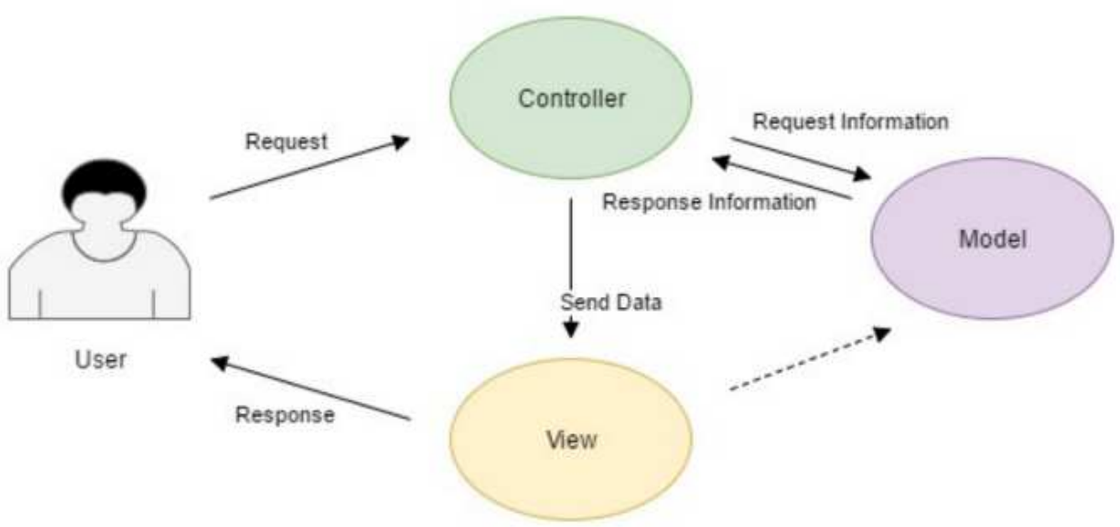
. . .

Explanation of MVC

To avoid these kinds of situations, developers started to think about a

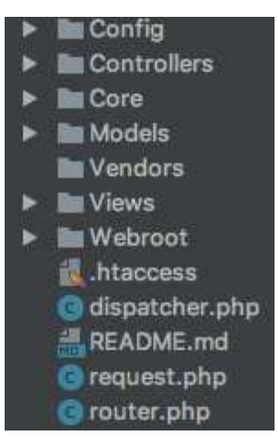
new way of organizing the code of a website. One way of doing it is the *MVC design pattern*. The goal is to divide the project into three big parts:

- **Model**: interacts with the database. It receives, stores and retrieves data for the user.
- **View**: displays information to the user and integrates data from the controller.
- **Controller**: sends and receives data from the model and passes to the view.



The structure

To set up this design pattern, we are going to structure our projects. We can find a lot of possibilities over the internet. But here is what I suggest:

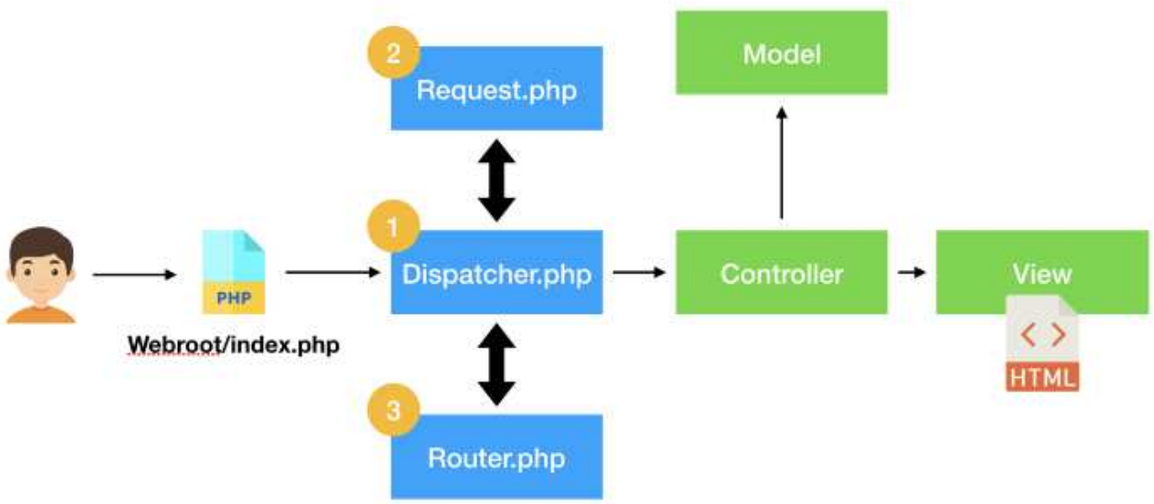


As you noticed, we retrieve the backbone of the MVC framework with

the three folders (Models, Views, Controllers) and some other stuffs :

- Webroot folder is the only directory accessible by the user.
- Router.php, dispatcher.php, request.php, .htaccess are part of the routing system
- Config : all the configuration needed by our website. We will retrieve then the db.php file which is the single access point to our database (singleton class).

Global Architecture



When accessing our website, the user will be automatically redirected to the Webroot/index.php thanks to two.htaccess files.

The first one will redirect the user to the Webroot directory.

```
RewriteEngine on
RewriteRule ^([a-zA-Z0-9\-\_\/]*)$ Webroot/$1
```

And the second one will redirect him/her to the index.php. Notice that we store the parameter (p=$1).

```
RewriteEngine on
RewriteCond %{REQUEST_URI} !\.(?:css|js|jpe?g|gif|png)$ [NC]
RewriteRule ^([a-zA-Z0-9\-\_\/]*)$ index.php?p=$1
```

The index.php is requiring all the files that we will need for the instantiation of the dispatcher. After creating a instance of the Dispatch class, we are ready to set our routing logic.

```php
define('WEBROOT', str_replace( search: "Webroot/index.php", replace: "", $_SERVER["SCRIPT_NAME"]));
define('ROOT', str_replace( search: "Webroot/index.php", replace: "", $_SERVER["SCRIPT_FILENAME"]));

require(ROOT . 'Config/core.php');

require(ROOT . 'router.php');
require(ROOT . 'request.php');
require(ROOT . 'dispatcher.php');

$dispatch = new Dispatcher();
$dispatch->dispatch();
```

Routing system

request.php

The goal of this file is to get the url requested by the user.

```php
class Request
{
    public $url;

    public function __construct()
    {
        $this->url = $_SERVER["REQUEST_URI"];
    }
}
```

router.php

The router takes the url captured by the *request.php* and explode the url into 3 different parts on the "/" character :

```php
$explode_url = explode( delimiter: '/', $url);
$explode_url = array_slice($explode_url, offset: 2);
$request->controller = $explode_url[0];
$request->action = $explode_url[1];
$request->params = array_slice($explode_url, offset: 2);
```

These inputs will be handled by the dispatcher. The dispatcher is doing the same job as an air traffic controller. When a new request is loaded, it selects the controller and the action with parameters. So with only one method (dispatch()), we can launch all this routing logic.

Database

Our model is going to handle the request to our database. So we will have to call our database a lot of time. In a simple way, at each connexion, we can create an instance of the database. This solution is not really efficient. I recommend we create a singleton which will handle the connexion to our database :

```php
class Database
{
    private static $bdd = null;

    private function __construct() {
    }

    public static function getBdd() {
        if(is_null(self::$bdd)) {
            self::$bdd = new PDO( dsn: "mysql:host=localhost;dbname=todo_PHP", username: 'root', passwd: 'root');
        }
        return self::$bdd;
    }
}
```

MVC

Now that we set up the dispatcher, our website can load an action from a controller.

Here we want to make a todo app, so we have to create a *tasksController.php*. This controller is going to ask for data from the model Task.php and then pass the data to a view. To make this process easier, we are going to create a parent class Controller that will handle this.

```php
class Controller
{
    var $vars = [];
    var $layout = "default";

    function set($d)
    {
        $this->vars = array_merge($this->vars, $d);
    }

    function render($filename)
    {
        extract($this->vars);
        ob_start();
        require(ROOT . "Views/" . ucfirst(str_replace( search: 'Controller', replace: '', get_class($this))) . '/' . $filename . '.php');
        $content_for_layout = ob_get_clean();

        if ($this->layout == false)
        {
            $content_for_layout;
        }
        else
        {
            require(ROOT . "Views/Layouts/" . $this->layout . '.php');
        }
    }
```

The *set()* method is going to merge all the data that we want to pass to the view.

The *render()* method is going to import the data with the method extract and then load the layout requested in the Views directory. Moreover, this allows us to have a layout in order to avoid the stupid repetition of HTML in our views.

We are ready to work on our tasksController.php. Just to test out our code, I'm going to create an index action :

And a quick view with a "Hello" message.

And here is the result :

Our MVC framework is set up ! Now we just have to make the CRUD actions about the task resource. If you want more details of this and get the website with the tasks CRUD, you can check out the repo on my

Github.

So now, you have developed an MVC structure that is a lot more sustainable than our traditional php website. But there is still a lot of work to do (security, error handling…). These topics are already handled by frameworks like Laravel or Symfony.

PHP Mvc Crud Router Database