معرفی میکروفریم ورکSlim3

تدوین: مهندس سمیرا احسانی

تاریخ آخرین ویرایش: 13 خرداد 1399

اداره سامانه های کاربردی – مرکز فاوای دانشگاه فردوسی مشهد

Slimدر واقع یک میکروفریم ورک بسیار کوچک و سبک است (در حد چند صد خط کد) که از امکانات آن می توان برای ساختن وب سایت های مختلف استفاده کرد .Slim در واقع همان قسمت (Controller) C در معماری MVC می باشد و چیزی بیش از یک کنترلر و dispatcher نیست. کاری که یک میکروفریم ورک انجام می دهد منحصرا این است که یک URL را دریافت می کند و آن را به یک عمل (Action) تبدیل می کند .

ایجاد و گسترش میکروفریم ورک ها از زمانی شروع شد که اکوسیستم PHP تغییر کرد. یکی از این تغییرات عمده استفاده از composer بود که سبب شد استفاده مجدد از کدها بسیار آسان شود. با استفاده از composer که یک package managerاست می توان یک فریم ورک و یا یک میکروفریم ورک که در واقع نقش کنترلر را دارد نصب کرد و سپس کدها و کتابخانه های دیگر را به راحتی به پروژه اضافه کرد) .استاندارد PSR-4 و یا استفاده از (autoloader

مشخصاتSlim3

توسط Josh Lockhartطراحی شده است که پایه گذار سایت www.phptherightway.com و نویسنده کتاب Modern PHP می باشد. (کتاب Modern PHP به صورت خلاصه امکانات جدید اضافه شده به PHP را خیلی ساده و با مثال توضیح می دهد. مطالعه این کتاب را به همه دوستان پیشنهاد می کنم).

Slim 3 مطابق با استاندارد PSR-7 طراحی شده است (PSR-7 استاندارد ارائه شده برای Http Response & Requestمی باشد.) بنابراین می توان از Middleware های سیستم های دیگر به راحتی در آن استفاده کرد. بر اساس معماری Middleware طراحی شده است که بیشتر به این مسئله خواهیم پرداخت.

در نهایت Slim3 دارای یک DIC (Dependency Injection Container) بسیار کوچک به نام pimpleمی باشد .

طریقه نصب و راه اندازی Slim 3

نصب composer

```
apt-get install composer
```

یک دایرکتوری با نام پروژه دلخواه خود) مثلا (myproject در Document rootوب سرور ایجاد کنید و در پنجره خط فرمان در مسیر آن قرار بگیرید و سپس با استفاده از دستور زیر Slim 3 را نصب کنید.

```
composer require slim/slim "^3.0"
```

برای شروع باید یک نقطه ورد به اپلیکیشن ایجاد کنیم برای این منظور باید یک فایل که در نقش front controllerباشد ایجاد کنیم مثلا یک فایل با نام index.phpدر دایرکتوری پروژه ایجاد می کنیم و کدهای زیر را در آن قرار می دهیم :

```
<?php
use \Psr\Http\Message\ServerRequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;
// Setup autoloader

require 'vendor/autoload.php';
use \Slim\App;

// Prepare app
$app = new App();
// Run app
$app->run();
```

اگر در مرورگر http://localhost/myproject/ را وارد کرده و اجرا کنیم خطای Page Not Found می دهد. همانطور که قبلا توضیح داده شد کار میکروفریم ورک ها تبدیل یک URL به یک action است حال باید ببینیم چگونه می توان به میکروفریم ورک گفت که برای URL خاصی عملیات مورد نظر ما را اجرا کند. برای این منظور باید route های مورد نظر خود را تعریف کرده، همراه با متد http آن و عملی که می خواهیم در صورت دریافت آن route اجرا شود به متغییر app اضافه کنیم.

```
$app->get('/', function (Request $request, Response $response) {
   $response->getBody()->write("Hello World");
return $response;
});
```

- در کد بالا get متد http است که URL با آن ارسال می شود و فقط درصورتی که URL با متد صحیح (در اینجا get) از جانب کلاینت ارسال شود تابعی که نوشته ایم اجرا خواهد شد. به جای get می توان از دیگر http verb ها مانند post, delete, put, patch, … هم استفاده کرد .

- '/'الگوی URL می باشد در اینجا با وارد کردن مسیر http://localhost/myproject/و تنظیم متد http بر روی get (از طریق ابزار postman) تابعی که در بالا نوشته ایم اجرا شده و Hello World بر روی صفحه نمایش داده خواهد شد .
- پارامتر دوم که در واقع یک تابع بی نام یا closure است. در صورت دریافت URL با الگوی تعریف شده در پارامتر اول و درست بودن متد http این تابع فراخوانی می شود و پاسخی را ایجاد کرده و آن را برمی گرداند .

به همین ترتیب می توان تمام route های مدنظر را به app اضافه کرد. منتها برای اینکه تمامی URL های ارسالی به سمت سرور به صفحه index.php فرستاده شوند؛ باید یک فایل .htaccess در دایرکتوری پروژه ایجاد کرده و همه route ها را به سمت صفحه index.php تغییر مسیر دهیم یا اصطلاحا rewrite کنیم:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [QSA,L]
```

نکته: Slim، مسیرها (rout ها)را به ترتیبی که در کد برنامه نوشته شده است بررسی و match می کند بنابراین route هایی که اختصاصی تر هستند باید قبل از route های عمومی تر قرار بگیرند. مثلا اگر دو مسیر ticket/new/و ticket/{id/} را داشته باشیم ابتدا باید مسیر اولی که اختصاصی تر است و سپس مسیر دومی که عمومی تر است را در کد قرار دهیم در غیر اینصورت همواره تابع مربوط به مسیر دوم اجرا می شود و Slim کلمه new را به عنوان مقدار برای پارامتر id در نظر می گیرد، به همین ترتیب می توان route های دینامیک همراه با آرگومان و یا با استفاده از عبارتهای منظم هم تعریف کرد. نام پارامترها را باید داخل {} قرار داد .

```
$app->get('/hello/{name}',function($request, $response, $args){
return $response->write("Hello " . $args['name']);
});
```

در کد بالا name پارامتری است که می توان برای آن مقادیر مختلف ارسال کرد مثلا وارد کرن آدرس http://localhost/myproject/hello/Aliخروجی Hello Aliرا برمی گرداند .در قطعه کد زیر از عبارات منظم برای تعریف الگوی route استفاده کرده ایم. در اینجا پارامتر name فقط می تواند شامل حروف الفبا باشد.

```
$app->get('/hello/{name[\w]+}', $Callable);
```

برای مشاهده مثالهای بیشتر در مورد نحوه تعریف route ها می توانید به نمونه های کد موجود در پروژه studentworkبر روی Git سرور مراجعه نمایید .با استفاده از دستور setName می توان به هر route یک نام اختصاص داد و سپس با استفاده از آن نام، آن متد را فراخوانی کرد :

```
$app->get('/hello/{name}',$Callable )->setName('hi');
```

در صورتی که به یک route نام اختصاص داده باشیم می توانیم با استفاده از متدURL ، urlForمورد نظر خود برای آن route خاص را بسازیم.

```
$link = $this->router->urlFor('hi', ['name' => 'Rob']);
```

با این روش اگر تغییری در قسمت تعریف route اعمال کنیم لازم نیست قسمت های دیگر کد را تغییر دهیم .

Middleware

Middlewareقطعه کدی است که بین Request و Response قرار می گیرد .Request را به عنوان ورودی دریافت کرده بر روی آن عملیاتی انجام می دهد و response را تولید نموده آن را برگردانده و یا به Middlewareبعدی در زنجیره Middlewareها پاس می دهد. مثلا می توان قسمت Authentication و Session را به صورت یک Middleware تعریف کرد که تمامی درخواست ها قبل از اینکه به App وارد شوند از این لایه عبور کنند و پاسخ تولید شده توسط APP هم پس از عبور از این کدهای میانی به دست کلاینت برسد .

امضای Middleware به این صورت است که سه متغییر ورودی دارد که $requestو $responseو $next می باشد که $next یک callable است که در واقع Middlewareبعدی در زنجیره Middleware ها و یا اگر در انتهای زنجیره باشد $appاست .

برای نوشتن کد Middleware چندین روش مختلف وجود دارد :

۱ -کد Middleware را در یک تابع بی نام نوشته و آن را در یک متغییر ذخیره کنیم.

```
$middleware = function ($request, $response, $next) {
   $response->getBody()->write('BEFORE');
   $response = $next($request, $response);
   $response->getBody()->write('AFTER');
   return $response;
}
$app->add($middleware);
```

۲ -کد Middleware را به صورت یک closure در همان محل استفاده بنویسیم.

```
$app->add(function ($request, $response, $next) {
   $response->getBody()->write('BEFORE');
   $response = $next($request, $response);
   $response->getBody()->write('AFTER');
   return $response;
});
```

۳ -یک کلاس با نام Middleware ایجاد کنیم و کد Middleware را در داخل تابع جادویی __invoke بنویسیم. در فایل: Middleware.php

```
class Middleware {
   public function __invoke($request, $response, $next)
   {
      $response->getBody()->write('BEFORE');
      $response = $next($request, $response);
      $response->getBody()->write('AFTER');
      return $response;
   }
}
```

در فایلIndex.php

```
$midware= new Middleware();
$app->add($midware);
```

برای اینکه Middleware ها یکی پس از دیگری اجرا شوند باید حتما شامل دستور زیر باشند.

```
$response = $next($request, $response);
```

اما اگر از cachingدر Middleware استفاده می کنید باید خط بالا را حذف کنید و return $response را قرار دهید زیرا در caching نیازی نیست که تمامی زنجیره پیموده شود و پاسخ از cache بازیابی می شود Slim. دارای Middlewareهای است که کار caching را انجام می دهند . می توان چندین Middleware را به صورت زنجیره ای به اپلیکیشن اضافه کرد .ترتیب اجرای Middleware ها به این صورت است که آخرین Middlewareدر زنجیره اولین Middleware است که اجرا می شود و به همین ترتیب اجرا در زنجیره به عقب برمی گردد تا به $app برسد.

```
$app->add($thirdMid)->add($secondMid)->add($firstMid);
```

چه قسمت هایی از کد بهتر است به صورت Middleware تعریف شوند؟

- در سطح Application
  - Authentication
  - Navigation
  - Session
- در سطحRoute
  - Access Control
  - Validation

Slimفریم ورک سبکی است و به صورت پیش فرض فقط شامل کنترولر می باشد برای افزودن قسمت View به اپلیکیشن گزینه های مختلفی وجود دارند. دوتا از مناسبترین آنها Twig و php-view می باشند که از طریق composer می توان آنها را به پروژه افزود. در مورد microservice ها چون در زمان نوشتن کد قسمت

کلاینت وجود ندارد برای تست کردن برنامه از ابزاری به نام postman استفاده می کنیم. این ابزار این امکان را به ما می دهد که request خود را با تنظیم http verbها و پارامترهای صفحه به سرور ارسال کنیم و microserviceدلخواه را فراخوانی کرده و نتیجه را مشاهده کنیم .

Slimمنطبق بر PSR-4 می باشد (autoloading) لذا می توان نگاشت namespace ها به دایرکتوری های موجود بر روی هارد را با افزودن کد زیر به composer.json انجام داد:

```
"autoload": {
"psr-4": { "Monolog\\": ["src/", "lib/"] }
```

دسته بندی route ها

در Slim می توان با استفاده از متد group مسیرها و یا همانroute های مشابه را دسته بندی کرد. این کار از لحاظ ساخته یافتگی و خوانایی کدها بهتر است و از طرفی این امکان را می دهد که یک middleware را به گروهی ازroute ها اضافه کنیم و لازم نیست که برای هر route جداگانه این کار را انجام بدهیم .

```
$app->group('/panel', function() use ($app) {
   $app->group('/admin', function() use ($app) {
      $app->get('', 'Admin/DashboardController:index');
      $app->get('/users', 'Admin/UserController:index');
      $app->post('/users', 'Admin/UserController:create');
   })->add($adminAuth);
})->add($userAuth);
```

در قطعه کد بالا تمامی route های panel در یک گروه دسته بندی شده اند و middlewareتعیین هویت کاربر userAuth بر روی آنها اعمال شده است علاوه بر این route های قسمت admin هم در زیر گروه دیگری دسته بندی شده و علاوه بر middlewareاولی middleware دیگری که مختص تعیین هویت مدیران است یعنی adminAuth هم بر روی آنها اعمال می شود .

(DIC (Dependency Injection Container:

Slimاز یک Dependency Container برای آماده کردن، مدیریت و تزریق Dependencyها به اپلیکیشن استفاده می کند. وقتی یک شیء از نوع App ایجاد می کنیم می توانیم به آن یک ورودی بدهیم این ورودی می تواند یک کانتینر و یا یک آرایه از settings باشد. اگر آرایه ارسال کنیم یک شیء از نوع کانتینر به صورت اتومات ایجاد شده و مقادیر آرایه در $container[‘settings’] ثبت می شود .

```
$container = new \Slim\Container;
$app = new \Slim\App ($container);
```

Dependency Injectionبه صورت بسیار کلی یعنی کاهش وابستگی بین کلاس ها از طریق وابسته کردن آنها به یک کلاس انتزاعی که به آن Interface می گویند .Interface در واقع یک اعلان کلاس هست همراه

با متغییرها و متدهای آن ولی شامل هیچ نوع پیاده سازی نمی شود. تمامی کلاس هایی که Interface خاصی را پیاده سازی می کنند در واقع از نوع آن Interface هم هستند و تمامی متدها و متغییرهای آن را نیز دارا می باشند. به عنوان مثال فرض کنید کلاسی با نام Dog داریم که از کلاس Canine ارث بری کرده است و از طرفی کلاسی به نام Parrot داریم که از کلاس Birds ارث بری کرده است. هر دوی این حیوانات جدا از تفاوت های کلی که با هم دارند می توانند نقطه اشتراکی داشته باشند مثلا هر دو می توانند به عنوان حیوان خانگی (Pet) نگهداری شوند. همه سگسانان و پرندگان را نمی توان به عنوان حیوان خانگی نگهداری کرد بنابراین متدها و متغییر های مربوط به حیوان خانگی بودن را نمی توان در کلاس والدین این دو قرار داد. در اینجا برای برقراری ارتباط بین دو کلاس که در ساختار منطقی ارث بری ارتباط والد و فرزندی و یا sibling بودن با هم ندارند از مفهومی به نام Interface استفاده می کنیم. در interface فقط متدها را اعلان می کنیم و سپس هر کدام از کلاس های Dog و Parrot آن را به شیوه مناسب برای خود پیاده سازی می کنند. نکته جالب اینجاست که هر شی از نوع Dog و یا Parrotشیء ای از نوع Pet هم به حساب می آید بنابراین اگر ما در کلاس دیگری متدی داشته باشیم که شیء ای از نوع Pet می پذیرد آنگاه می توانیم هر شیء ای از نوع Dog و یا Parrot را به عنوان Petبه آن ارسال کنیم. این نوع طراحی باعث می شود که وابستگی کلاس ها به یکدیگر کمتر شود .مسئله دیگر در Dependency Injection این است که اگر در کلاس A شیء از کلاس B ایجاد کرده و در یک متغییر کلاس A قرار بدهیم همانند کد زیر:

```
$this->b = new B();
```

به جای اینکه شیء را در داخل متدی از کلاس A که آن را نیاز دارد ایجاد کنیم آن را به عنوان ورودی به سازنده کلاس Aارسال کنیم تا در هنگام ایجاد شیء از کلاسA ؛ شیء ای که کلاس A به آن وابسته است یعنی شیء از نوع کلاس B از قبل ایجاد شده باشد.

```
class A {
   pubic $b;
   public function __construct( B $b){
      $this->b = $b;
   }
   public function getB(){
      //$this->b = new B();  //Don't do it!!!
      return $this->b;
   }
}
```

کاری که Dependency Injection Container انجام می دهد این است که اجازه مدیریت کردن و تزریق وابستگی ها را می دهد. کانتینر در واقع ظرفی است که در آن می توان متغییر ها و سرویس هایی را ذخیره کرد تا به صورت گلوبال در فضای اپلیکیشن استفاده شوند. مثلا می توان برای اتصال به دیتابیس و ایجاد یک شیء از نوع کلاس pdo که توسط سایر کلاس ها مورد استفاده قرار می گیرد سروریسی به صورت زیر ایجاد و در کانتینر ثبت کرد.

```
// Database connection
```

```
$container['pdo'] = function ($c) {
   $_host = 'localhost';
   $_user = 'root';
   $_pass = 'root';
   $_default_db ='test';
   $pdo = new PDO("mysql:host=" . $_host . ";dbname=" . $_default_db, $_user, $_pass,
   array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"));
   return $pdo;
};
```

می توان به صورت زیر به سروریسی که در بالا تعریف کرده ایم دسترسی داشته باشیم:

```
$this->get('pdo');
$this->pdo;
```

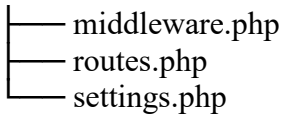
هر جا که به این صورت سرویس pdo را فراخوانی کنیم اگر شیء از کلاس PDO ایجاد نشده باشد آن را ایجاد می کند و برمی گرداند ولی اگر موجود باشد همان شیء قبلی را برمی گرداند .نحوه مدیریتdependency ها با استفاده از ثبت سرویس در کانتینر و با طرح یک مشکل و با مثال در اینجا توضیح داده شده است. لطفا برای مطالعه بیشتر به آن رجوع کنید .

ساختار پیشنهادی برای سازماندهی فایلها و دایرکتوری های پروژه

```
/
├──app/
├──cache/
├──public/
│   ├── css/
│   ├── js/
│   └── index.php
├──vendor/
├──composer.json
└──composer.lock
```

شاخه app شامل کدهای پروژه می باشد که به صورت نمونه می تواند به شکل زیر باشد.

```
app/
├── src/
│   ├── App/
│   ├── Photos/
│   │   ├── FlickrService.php
│   │   └── Photo.php
├── templates/
│   ├── layout.twig
│   └── app/
│      └── home/
│         └── list.twig
├── dependencies.php
```


middleware.php
routes.php
settings.php