30.08.2020

Современный PHP без использования фреймворка

У меня есть для вас вызов. В следующий раз, когда вы начнёте новый проект, попробуйте не использовать PHP-фреймворк.

Это не попытка критики фреймворков. Также это не призыв к отказу от использования готовых решений. В этом уроке мы будем использовать некоторые пакеты, разработанные разработчиками фреймворков. У меня есть огромное уважение к инновациям, которые происходят в этой области.

Здесь речь идет не о них. Здесь речь идет о вас. Речь идет о том, чтобы дать себе возможность расти как разработчику.

Одним из наиболее значимых преимуществ работы без фреймворка является глубокое понимание происходящего "под капотом". Вы сможете увидеть точно, что происходит, не полагаясь на магию фреймворка, которую нельзя отлаживать и которую вы на самом деле не совсем понимаете.

Очень вероятно, что ваша следующая работа не предоставит возможности начать зеленое поле проекта с выбранным вами фреймворком. Реальность заключается в том, что большинство важных бизнес-задач на PHP связаны с существующими приложениями. Независимо от того, построено ли это приложение на популярно поддерживаемом фреймворке, таком как Laravel или Symfony, фреймворке прошлых дней, таком как CodeIgniter или FuelPHP, или даже старомодном PHP-приложении с "архитектурой на основе include", разработка без фреймворка сейчас поможет вам лучше подготовиться к выполнению любого проекта на PHP в будущем.

В прошлом было непросто создавать приложения без фреймворка, так как требовалась система, способная интерпретировать и маршрутизировать HTTP-запросы, отправлять HTTP-ответы и управлять зависимостями. Отсутствие отраслевых стандартов означало, что по крайней мере эти компоненты фреймворка были тесно связаны. Если вы не использовали фреймворк, вам приходилось создавать его самостоятельно.

Однако сегодня, благодаря работе PHP-FIG по автозагрузке и взаимодействию, создание приложения без фреймворка не означает создание всего с нуля. Существует множество отличных пакетов, совместимых между собой, разработанных различными поставщиками. Собрать все воедино гораздо проще, чем вы думаете!

Как работает PHP

Прежде чем перейти к чему-либо еще, важно понять, как взаимодействуют PHP-приложения с внешним миром.

PHP выполняет серверные приложения по принципу цикла запроса/ответа. Каждое взаимодействие с вашим приложением - будь то через браузер, командную строку или REST API - поступает в приложение в виде запроса. Когда запрос получен, приложение запускается, обрабатывает запрос для генерации ответа, ответ отправляется обратно клиенту, сделавшему запрос, и приложение завершает работу. Это происходит при каждом взаимодействии.

The front controller

Вооружившись этим знанием, мы начнем с фронт-контроллера. Фронт-контроллер - это PHP-файл, который обрабатывает каждый запрос для вашего приложения. Это первый PHP-файл, с которым запрос сталкивается при входе в ваше приложение и, по сути, последний PHP-файл, через который проходит ответ при выходе из вашего приложения.

Давайте используем классический пример "Привет, мир!", который будет предоставлен встроенным веб-сервером PHP, чтобы убедиться, что все подключено правильно. Если вы еще не установили, убедитесь, что у вас установлена PHP версии 7.2 или новее в вашей среде разработки.

Создайте директорию проекта с вложенной директорией "public", а затем внутри нее создайте файл index.php со следующим кодом.

```
<?php
declare(strict_types=1);

echo 'Hello, world!';
```

Обратите внимание, что здесь мы объявляем строгую типизацию (strict typing), что является хорошей практикой и следует делать в начале каждого PHP-файла вашего приложения. Типизация помогает при отладке и ясно выражает намерения разработчика, который будет работать с кодом в будущем.

Перейдите в директорию вашего проекта, используя командную строку (например, Терминал на macOS), и запустите встроенный веб-сервер PHP.

```
php -S localhost:8080 -t public/
```

Теперь загрузите http://localhost:8080/ в вашем браузере. Вы видите "Привет, мир!" без каких-либо ошибок?

Отлично! Теперь перейдем к основной части!

Автозагрузка и сторонние пакеты

Когда вы только начинали работать с PHP, возможно, вы использовали операторы include или require во всем вашем приложении, чтобы включить функциональность или конфигурацию из других PHP-файлов. В целом, мы хотим избегать такого подхода, потому что это затрудняет понимание пути выполнения кода и определение зависимостей. Это может привести к сложностям при отладке.

Мы могли бы написать свой собственный автозагрузчик, но поскольку мы собираемся использовать Composer для управления сторонними зависимостями, и он уже включает в себя надежный автозагрузчик, давайте просто воспользуемся им.

Убедитесь, что у вас установлен Composer на вашей системе. Затем настройте Composer для данного проекта.

```
composer init
```

Это позволит вам пройти интерактивное руководство для создания файла конфигурации composer.json. После завершения этого процесса, откройте файл в редакторе и добавьте поле autoload, чтобы он выглядел примерно так (Это гарантирует, что автозагрузчик знает, где искать классы нашего приложения.)

Теперь установите Composer для этого проекта, что позволит подключить все зависимости (если они уже есть) и настроить автозагрузчик для нас.

Выполните следующую команду в командной строке, находясь в корневой директории проекта:

```
composer install
```

Обновите файл public/index.php, чтобы подключить автозагрузчик. Идеально, это будет один из немногих операторов "include", которые вы будете использовать в вашем приложении.

```
<?php
declare(strict_types=1);

require_once dirname(__DIR__) . '/vendor/autoload.php';
echo 'Hello, world!';
```

Если вы перезагрузите ваше приложение в браузере, вы не заметите никаких изменений. Автозагрузчик присутствует, он просто пока не выполняет никакой серьезной работы для нас. Давайте переместим пример "Hello, world!" в автозагружаемый класс, чтобы увидеть, как это работает.

Создайте новую директорию из корневой директории проекта с именем src, а затем внутри нее добавьте файл HelloWorld.php со следующим кодом.

```
<?php
declare(strict_types=1);

namespace ExampleApp;

class HelloWorld
{
   public function announce(): void
   {
      echo 'Hello, autoloaded world!';
   }
}
```

Now in public/index.php, replace the echo statement with a call to the announce method on the HelloWorld class.

Теперь в public/index.php замените оператор echo вызовом метода announce класса HelloWorld.

```
// ...

require_once dirname(__DIR__) . '/vendor/autoload.php';

$helloWorld = new \ExampleApp\HelloWorld();$helloWorld->announce();
```

Перезагрузите ваше приложение в браузере, чтобы увидеть новое сообщение!

Контейнер зависимостей

Dependency Injection (DI) - это программная техника, при которой каждая зависимость предоставляется классу, который ее требует, вместо того чтобы класс самостоятельно обращался за необходимой информацией или функциональностью во внешнюю среду.

Например, предположим, что метод класса в вашем приложении должен выполнять чтение из базы данных. Для этого вам понадобится подключение к базе данных. Обычным подходом является создание нового подключения с учетными данными, полученными из глобальной области.

```
class AwesomeClass
{
   public function doSomethingAwesome()
   {
      $dbConnection = new \PDO(
         "{$_ENV['type']}:host={$_ENV['host']};dbname={$_ENV['name']}",
         $_ENV['user'],
         $_ENV['pass']
      );

      // Make magic happen with $dbConnection
   }
}
```

Однако такой подход запутанный, поскольку он наделяет этот метод ответственностями, которые на самом деле не относятся к нему - создание нового объекта подключения к базе данных, получение учетных данных и обработка возможных проблем, которые могут возникнуть, если подключение не удалось. Кроме того, это приводит к дублированию кода по всему приложению. Если вы попытаетесь протестировать этот класс, вы обнаружите, что это невозможно. Этот класс сильно связан как с окружением приложения, так и с базой данных.

Вместо этого, почему бы не быть как можно более ясным в отношении того, что вашему классу вообще нужно? Давайте просто потребуем, чтобы объект PDO был внедрен в класс изначально.

```
class AwesomeClass
{
   private $dbConnection;
   public function __construct(\PDO $dbConnection) {
      $this->dbConnection = $dbConnection;
   }
   public function doSomethingAwesome()
   {
      // магия $this->dbConnection
   }
}
```

Это гораздо более чистый, понятный и менее подверженный ошибкам подход. Благодаря подсказке типа и внедрению зависимостей, метод объявляет точно то, что ему нужно для выполнения своей работы, и получает это без вызова внешней зависимости изнутри самого себя. Когда дело доходит до модульного тестирования, у нас есть отличная возможность имитировать подключение к базе данных и передать его в метод.

Теперь контейнер внедрения зависимостей - это инструмент, который вы оборачиваете вокруг всего вашего приложения, чтобы обрабатывать создание и внедрение этих зависимостей. Контейнер не обязательно требуется для использования техники внедрения зависимостей, но он значительно облегчает работу с ростом и сложностью вашего приложения.

Мы будем использовать один из самых популярных контейнеров внедрения зависимостей для PHP - PHP-DI, который имеет подходящее название. (Стоит отметить, что его документация предлагает другой способ объяснения внедрения зависимостей, который может быть полезен для некоторых читателей.)

Контейнер внедрения зависимостей (Dependency Injection Container)

Теперь, когда у нас есть настроенный Composer, установка PHP-DI довольно проста. Вернитесь в командную строку, чтобы установить его.

```
composer require php-di/php-di
```

Обновите файл public/index.php для настройки и создания контейнера.

```
// ...

require_once dirname(__DIR__) . '/vendor/autoload.php';

$containerBuilder = new \DI\ContainerBuilder();
$containerBuilder->useAutowiring(false);
$containerBuilder->useAnnotations(false);
$containerBuilder->addDefinitions([
   \ExampleApp\HelloWorld::class => \DI\create(\ExampleApp\HelloWorld::class)
]);
$container = $containerBuilder->build();
$helloWorld = $container->get(\ExampleApp\HelloWorld::class);
$helloWorld->announce();
```

Пока что ничего особенного не происходит. Это все еще простой пример, где все находится в одном файле, чтобы было легко следить за происходящим.

Пока что мы просто настраиваем контейнер так, чтобы мы явно объявляли зависимости (вместо использования автоматической проводки или аннотаций) и извлекаем объект HelloWorld из контейнера.

Краткое отступление: Автоматическая проводка - это отличная функция, предлагаемая многими контейнерами внедрения зависимостей, которая помогает минимизировать бессмысленную настройку, но мы отключаем ее в этом руководстве, чтобы максимально улучшить ваше понимание. Явная настройка всех зависимостей здесь даст вам гораздо лучшее представление о том, что делает автоматическая проводка контейнера внедрения зависимостей "под капотом".

Давайте сделаем код еще более читабельным, импортировав пространства имен, где это возможно.

```
<?php
declare(strict_types=1);

require_once dirname(__DIR__) . '/vendor/autoload.php';

use DI\ContainerBuilder;
use ExampleApp\HelloWorld;
use function DI\create;

$containerBuilder = new ContainerBuilder();
$containerBuilder->useAutowiring(false);
$containerBuilder->useAnnotations(false);
$containerBuilder->addDefinitions([
    HelloWorld::class => create(HelloWorld::class)]);

$container = $containerBuilder->build();

$helloWorld = $container->get(HelloWorld::class);
$helloWorld->announce();
```

На данный момент это может показаться просто лишней суетой, чтобы делать то, что мы уже делали раньше.

Не волнуйтесь, контейнер пригодится, когда мы добавим несколько других инструментов, чтобы направлять запрос через наше приложение. Они будут использовать контейнер для загрузки нужных классов в нужное время и место.

Middleware

Если представить ваше приложение как лук, в котором запросы приходят снаружи, проходят через центр лука и возвращаются обратно в виде ответов, то промежуточное программное обеспечение (middleware) представляет собой каждый слой лука, который получает запрос, потенциально выполняет какие-то действия с этим запросом и либо передает его на следующий уровень, либо создает ответ и отправляет его обратно на предыдущий уровень (это может произойти, если middleware проверяет определенное условие, которое не выполняется, например, запрос несуществующего маршрута).

Если запрос успешно проходит через все слои, приложение обрабатывает его и преобразует в ответ, и каждое промежуточное программное обеспечение в обратном порядке получает ответ, потенциально изменяет его и передает следующему промежуточному программному обеспечению.

Промежуточное программное обеспечение может быть полезным во множестве сценариев, таких как:

Отладка проблем во время разработки
Гармоничная обработка исключений в продакшн
Ограничение количества входящих запросов (rate-limiting)
Обработка входящих запросов с неподдерживаемыми типами медиа
Обработка CORS
Маршрутизация запросов к соответствующему классу обработчику

Можно ли использовать только промежуточное программное обеспечение для реализации инструментов для обработки этих сценариев? Конечно, нет. Однако, использование промежуточного программного обеспечения делает цикл запроса/ответа более ясным, что облегчает отладку и ускоряет разработку.

Мы воспользуемся промежуточным программным обеспечением для последнего указанного сценария: маршрутизации.

Роутинг

Маршрутизатор использует информацию из входящего запроса, чтобы определить, какой класс должен его обработать. (Например, URI /products/purple-dress/medium должен быть обработан классом ProductDetails с передачей purple-dress и medium в качестве аргументов.)

В нашем примере приложения мы будем использовать популярный маршрутизатор FastRoute через реализацию промежуточного программного обеспечения, совместимого с PSR-15.

The middleware dispatcher

Чтобы наше приложение работало с промежуточным программным обеспечением FastRoute (и любым другим промежуточным программным обеспечением, которое мы установим), нам понадобится диспетчер промежуточного программного обеспечения.

PSR-15 - это стандарт промежуточного программного обеспечения, который определяет интерфейсы для промежуточного программного обеспечения и диспетчеров (в спецификации они называются "обработчиками запросов"), обеспечивая совместимость между различными промежуточными программными обеспечениями и диспетчерами. Нам просто нужно выбрать совместимый с PSR-15 диспетчер, и мы можем быть уверены, что он будет работать с любым промежуточным программным обеспечением, совместимым с PSR-15.

Давайте установим Relay в качестве диспетчера. Он прост в использовании и не требует никакой конфигурации.

```
composer require relay/relay
```

И поскольку спецификация промежуточного программного обеспечения PSR-15 требует, чтобы реализации передавали совместимые с PSR-7 HTTP-сообщения, мы будем использовать Laminas Diactoros в качестве реализации PSR-7.

```
composer require laminas/laminas-diactoros
```

Давайте подготовим Relay для работы с промежуточным программным обеспечением.

```
// ...

use DI\ContainerBuilder;
use ExampleApp\HelloWorld;
use Relay\Relay;use Laminas\Diactoros\ServerRequestFactory;use function DI\create;

// ...

$container = $containerBuilder->build();

$middlewareQueue = [];
$requestHandler = new Relay($middlewareQueue);
$requestHandler->handle(ServerRequestFactory::fromGlobals());
```

Мы используем ServerRequestFactory::fromGlobals() на строке 16, чтобы собрать всю необходимую информацию для создания нового объекта Request и передать его в Relay. Здесь Request входит в наш стек промежуточного программного обеспечения.

Теперь давайте добавим промежуточное программное обеспечение FastRoute и обработчика запросов (request handler). (FastRoute определяет, является ли запрос допустимым и может ли быть обработан приложением, а обработчик запросов отправляет Request обработчику, настроенному для этого маршрута в определении маршрутов.)

composer require middlewares/fast-route middlewares/request-handler

И определим маршрут к нашему классу-обработчику Hello, world!. Здесь мы используем маршрут /hello, чтобы показать, что маршрут отличный от базового URI работает.

```
// ...

use DI\ContainerBuilder;
use ExampleApp\HelloWorld;
use FastRoute\RouteCollector;
use Middlewares\FastRoute;
use Middlewares\RequestHandler;
use Relay\Relay;
use Laminas\Diactoros\ServerRequestFactory;
use function DI\create;
use function FastRoute\simpleDispatcher;
// ...

$container = $containerBuilder->build();

$routes = simpleDispatcher(function (RouteCollector $r) {
    $r->get('/hello', HelloWorld::class);
});
$middlewareQueue[] = new FastRoute($routes);
$middlewareQueue[] = new RequestHandler();
$requestHandler = new Relay($middlewareQueue);
$requestHandler->handle(ServerRequestFactory::fromGlobals());
```

Для того, чтобы это работало, вам также нужно обновить класс HelloWorld, чтобы он стал вызываемым классом, то есть класс может быть вызван, как если бы он был функцией.

```
// ...

class HelloWorld
{
    public function __invoke(): void    {
        echo 'Hello, autoloaded world!';
        exit;
    }
}
```

Обратите внимание на добавленный код exit; в магическом методе __invoke(). Мы скоро разберемся с этим — просто не хотел, чтобы вы его упустили.

Теперь загрузите http://localhost:8080/hello и наслаждайтесь своим успехом!

Клей, который объединяет все воедино

Дотошный читатель быстро заметит, что хотя мы все еще заботимся о настройке и создании контейнера внедрения зависимостей (DI), на самом деле он ничего не делает для нас. Диспетчер и промежуточное программное обеспечение могут выполнять свою работу без него.

Так когда же он вступает в игру?

Что, если - как это обычно бывает в реальном приложении - у класса HelloWorld есть зависимость?

Давайте введем незначительную зависимость и посмотрим, что произойдет.

```
// ...

class HelloWorld
{
    private $foo;
    public function __construct(string $foo) {
        $this->foo = $foo;
    }

    public function __invoke(): void
    {
        echo "Hello, {$this->foo} world!";
        exit;
    }
}
```

Перезагрузите браузер, и...

Ого.

Обратите внимание на эту ошибку ArgumentCountError.

Это происходит потому, что теперь HelloWorld требует внедрения строки при создании, чтобы выполнить свою работу, и этот параметр не был передан. Здесь контейнер помогает.

Давайте определим эту зависимость в контейнере и передадим контейнер в RequestHandler, чтобы он мог ее разрешить.

```
// ...

use Laminas\Diactoros\ServerRequestFactory;
use function DI\create;
use function DI\get;
use function FastRoute\simpleDispatcher;

// ...

$containerBuilder->addDefinitions([
    HelloWorld::class => create(HelloWorld::class)
        ->constructor(get('Foo')), 'Foo' => 'bar']);

$container = $containerBuilder->build();

// ...

$middlewareQueue[] = new FastRoute($routes);
$middlewareQueue[] = new RequestHandler($container);
$requestHandler = new Relay($middlewareQueue);
$requestHandler->handle(ServerRequestFactory::fromGlobals());
```

Ура! Вы должны увидеть Hello, bar world! при обновлении страницы в браузере.

Правильная отправка ответов

Помните ранее, когда я упомянул оператор выхода exit в классе HelloWorld?

Это быстрый и грубый способ гарантировать простой ответ в процессе разработки, но это не лучший способ отправки вывода в браузер. Такая грубая техника заставляет HelloWorld выполнять дополнительную работу по отправке ответа, что на самом деле должно быть ответственностью другого класса. Это усложняет отправку правильных заголовков и кодов состояния, и завершает приложение, не давая возможности выполнить промежуточное программное обеспечение, следующее после HelloWorld.

Помните, что каждый промежуточный слой имеет возможность изменить Request при входе в наше приложение и (в обратном порядке) изменить ответ при выходе из приложения. Кроме общего интерфейса для Request, PSR-7 также определяет структуру для другого HTTP-сообщения, которое поможет нам во второй половине этого цикла: Response. (Если вы действительно хотите погрузиться в детали, прочитайте все о HTTP-сообщениях и о том, что делает стандарты PSR-7 Request и Response настолько замечательными.)

Обновите класс HelloWorld, чтобы он возвращал Response.

```
// ...

namespace ExampleApp;

use Psr\Http\Message\ResponseInterface;
class HelloWorld
{
    private $foo;

    private $response;
    public function __construct(
        string $foo,
        ResponseInterface $response
    ) {
        $this->foo = $foo;
        $this->response = $response;
    }

    public function __invoke(): ResponseInterface {
        $response = $this->response->withHeader('Content-Type', 'text/html');
        $response->getBody()
            ->write("<html><head></head><body>Hello, {$this->foo} world!</body></html>");
            return $response;
    }}
```

И обновите определение контейнера, чтобы предоставить классу HelloWorld новый объект Response.

```
// ...

use Middlewares\RequestHandler;
use Relay\Relay;
use Laminas\Diactoros\Response;
use Laminas\Diactoros\ServerRequestFactory;
use function DI\create;

// ...

$containerBuilder->addDefinitions([
    HelloWorld::class => create(HelloWorld::class)
```

```
        ->constructor(get('Foo'), get('Response')), 'Foo' => 'bar', 'Response' => function() {
            return new Response();
        }
]);

$container = $containerBuilder->build();

// ...
```

Если вы сейчас перезагрузите страницу, то получите пустой экран. Наше приложение возвращает правильный объект Response от диспетчера middleware, но что дальше?

Оно ничего не делает.

Нам нужен еще один инструмент, чтобы завершить всю работу: эмиттер. Эмиттер находится между вашим приложением и веб-сервером (Apache, nginx и т. д.), который будет отправлять ваш ответ клиенту, инициировавшему запрос. Он фактически берет объект Response и преобразует его в инструкции, которые может понять серверное API.

Давайте подключим Narrowspark's HTTP Emitter.

```
composer require narrowspark/http-emitter
```

Обновите файл public/index.php, чтобы получить объект Response от диспетчера и передать его эмиттеру.

```
// ...

use Middlewares\FastRoute;
use Middlewares\RequestHandler;
use Narrowspark\HttpEmitter\SapiEmitter;
use Relay\Relay;
use Laminas\Diactoros\Response;

// ...

$requestHandler = new Relay($middlewareQueue);
$response = $requestHandler->handle(ServerRequestFactory::fromGlobals());
$emitter = new SapiEmitter();
return $emitter->emit($response);
```

Перезагрузите браузер, и мы снова в бизнесе! И на этот раз у нас гораздо более надежный способ обработки ответов.

В строке 15 в приведенном выше коде заканчивается цикл запроса/ответа в нашем приложении, и вступает в действие веб-сервер.

Обратите внимание, что в данном примере конфигурация эмиттера очень простая. Хотя она может быть более сложной, реальное приложение должно быть настроено на автоматическое использование эмиттера потокового вывода для больших загрузок. В документации Laminas описывается интересный способ реализации этого с помощью HTTP Request Handler Runner, комбинированного диспетчера PSR-15 и эмиттера PSR-7.

Заключение

Вот и всё! С помощью всего 44 строк кода и нескольких широко используемых, тщательно протестированных и надежных компонентов, мы создали основу для современного PHP-приложения. Оно соответствует стандартам PSR-4, PSR-7, PSR-11 и PSR-15, что означает, что вы можете выбрать любую из множества реализаций этих стандартов от разных поставщиков для работы с HTTP-сообщениями,

контейнером внедрения зависимостей, промежуточным программным обеспечением и диспетчером промежуточного программного обеспечения.

Мы углубились в некоторые технологии и объяснили принципы, на которых базируются наши решения. Я надеюсь, что теперь вы видите, насколько просто создать новое приложение без лишней сложности фреймворка. И, что еще более важно, я надеюсь, что вы теперь готовы использовать эти техники в существующем приложении, когда возникнет необходимость.

Если вы ищете ещё больше отличных и качественных пакетов, которые легко интегрировать, я с уверенностью рекомендую обратить внимание на следующие проекты: Aura, The League of Extraordinary Packages, компоненты Symfony, компоненты Laminas, библиотеки, ориентированные на безопасность, от Paragon Initiative, а также этот список промежуточного программного обеспечения, соответствующего стандарту PSR-15.

Если вы планируете использовать этот пример кода в продакшн-среде, вероятно, вам будет удобно вынести определения маршрутов и контейнера в отдельные файлы, чтобы облегчить их поддержку по мере роста сложности. Также рекомендуется реализовать EmitterStack для умной обработки загрузок файлов и других больших ответов.