ПРОГРАММНЫЙ КОМПЛЕКС «UDS FRAMEWORK»

Инструкция по эксплуатации

2023

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

1.1. Область применения

Программный комплекс «UDS Framework» используется как программная платформа для разработки WEB-приложений.

1.2. Краткое описание возможностей

Программный комплекс «UDS Framework» предназначен для решения следующих задач:

1) создание WEB-приложений от минимальных сервисов до крупных приложений;
2) поддержка различных СУБД. Легкая расширяемость позволяет с минимальными трудозатратами интегрировать любую реляционную СУБД;
3) расширяемость кода сторонними библиотеками благодаря использованию модульной системы и менеджера пакетов Composer;
4) интегрирование сторонних API путем добавления новых сервисов;
5) возможность создания как API, так и цельных WEB-приложений с использованием шаблонизаторов;
6) возможность полного логирования всех запросов, действий, ответов благодаря интегрированной системе событий;
7) возможность быстрого и удобного администрирования программы за счёт использования технологий Docker-контейнеров.

1.3. Преимущества

Основным преимуществом Программного комплекса является гибкость и масштабируемость за счёт использования методологии DI контейнера, а также интеграция компонентов с помощью предконфигурирования по рецептам.

1.4. Основные модули

- Flex – модуль предконфигурирования компонентов;
- Роутинг;
- DI – контейнер;
- Yaml – конфигурация;
- интеграция с ORM, что позволяет переключаться к любому типу баз данных;
- Модуль миграций баз данных;
- Секьюрити.

2. РАБОТА С «UDS FRAMEWORK»

2.1. Маршрутизация

Когда ваше приложение получает запрос, оно вызывает действие контроллера для генерации ответа. Конфигурация маршрутизации определяет, какое действие выполнять для каждого входящего URL. Он также предоставляет другие полезные функции, такие как генерация URL-адресов, удобных для SEO.

2.1.1. Создание маршрутов

Маршруты могут быть настроены в YAM, PHP или с использованием атрибутов. Оба варианта обеспечивают одинаковые функции и производительность, поэтому выбирайте то что больше нравится. Рекомендуется использовать атрибуты, потому что более удобно размещать маршрут и контроллер в одном и том же месте.

2.1.1.1 Создание маршрутов в качестве атрибутов

Атрибуты PHP позволяют определять маршруты рядом с кодом контроллеров, связанных с этими маршрутами. Атрибуты являются собственными в PHP 8 и более поздних версиях, поэтому вы можете использовать их сразу.

Для использования атрибутов добавьте в проект следующий фрагмент (Рисунок 1).

```yaml
# config/routes/attributes.yaml
controllers:
    resource:
        path: ../../src/Controller/
        namespace: App\Controller
    type: attribute
```

Рисунок 1 – Пример использования атрибутов

Эта конфигурация ищет маршруты, определенные как атрибуты в классах, объявленных в пространстве имен **App\Controller** и сохраненных в каталоге **src/Controller/**, который соответствует стандарту PSR-4, при этом ядро также может выступать в качестве контроллера.

Для того чтобы определить маршрут для URL-адреса **/blog** в приложении необходимо создать класс контроллера, подобный следующему (Рисунок 2).

```php
class BlogController extends AbstractController
{
    #[Route('/blog', name: 'blog_list')]
    public function list(): Response
    {
        // ...
    }
}
```

Рисунок 2 – Создание класса контроллера

2.1.1.2 Создание маршрутов в файлах YAML, XML или PHP

Вместо определения маршрутов в классах контроллеров, допускается их определение в отдельном файле YAML или PHP. Главное преимущество заключается в том, что при этом не требуется какой-либо дополнительной зависимости. Однако, при этом приходится работать с несколькими файлами при проверке маршрутизации какого-либо действия контроллера.

В следующем примере (Рисунок 3) показано, как определить маршрут с именем **blog_list**, который связывает URL-адрес **/blog** с действием **list() BlogController**:

```yaml
# config/routes.yaml
blog_list:
    path: /blog
    # the controller value has the format 'controller_class::method_name'
    controller: App\Controller\BlogController::list

    # if the action is implemented as the __invoke() method of the
    # controller class, you can skip the '::method_name' part:
    # controller: App\Controller\BlogController
```

Рисунок 3 – Пример определения маршрута

2.1.2. Совпадающие HTTP-методы

По умолчанию маршруты соответствуют любому HTTP-запросу (GET, POST, PUT и т.д.). Используйте параметр **methods**, чтобы ограничить запросы, на которые должен отвечать каждый маршрут (Рисунок 4)

```php
// src/Controller/BlogApiController.php
namespace App\Controller;

// ...

class BlogApiController extends AbstractController
{
    #[Route('/api/posts/{id}', methods: ['GET', 'HEAD'])]
    public function show(int $id): Response
    {
        // ... return a JSON response with the post
    }

    #[Route('/api/posts/{id}', methods: ['PUT'])]
    public function edit(int $id): Response
    {
        // ... edit a post
    }
}
```

Рисунок 4 – Пример ограничения запросов параметром **methods**

2.1.3. Параметры маршрута

В предыдущих примерах были определены маршруты, в которых URL-адрес никогда не меняется (например, /**blog**). Однако обычно определяются маршруты, где некоторые части являются переменными. Например, URL-адрес для отображения какой-либо записи в блоге, вероятно, будет включать заголовок (например, **/blog/my-first-post** или **/blog /all-about-data**).

В маршрутах переменные части заключаются в { }. Например, маршрут для отображения содержимого записи в блоге определяется как **/blog/{slug}** (Рисунок 5).

```php
class BlogController extends AbstractController
{
    // ...

    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show(string $slug): Response
    {
        // $slug will equal the dynamic part of the URL
        // e.g. at /blog/yay-routing, then $slug='yay-routing'

        // ...
    }
}
```

Рисунок 5 – Пример переменной части в маршруте

2.1.4. Преобразование параметров

Обычной необходимостью маршрутизации является преобразование значения, хранящегося в некотором параметре (например, целое число, действующее как идентификатор пользователя), в другое значение (например, объект, представляющий пользователя). Эта функция называется «преобразователем параметров». Для преобразования параметров маршрута **$slug** добавьте **BlogPost $post** (Рисунок 6).

```php
class BlogController extends AbstractController
{
    // ...

    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show(BlogPost $post): Response
    {
        // $post is the object whose slug matches the routing parameter

        // ...
    }
}
```

Рисунок 6 – Пример преобразования параметров маршрута

2.2. События и Прослушиватели событий

Во время выполнения приложения запускается множество уведомлений о событиях. Ваше приложение может прослушивать эти уведомления и отвечать на них, выполняя любой фрагмент кода.

UDS запускает несколько событий, связанных с ядром, во время обработки HTTP-запроса. Пакеты сторонних производителей также могут отправлять события, и вы даже можете отправлять пользовательские события из своего собственного кода.

Все примеры, показанные в этой статье, используют одно и то же событие **KernelEvents::EXCEPTION** для целей согласованности. В своем собственном приложении вы можете использовать любое событие и даже смешивать несколько из них в одном подписчике.

2.2.1. Создание прослушивателя событий

Наиболее распространенным способом прослушивания события является регистрация прослушивателя событий (Рисунок 7).

```php
class ExceptionListener
{
    public function onKernelException(ExceptionEvent $event)
    {
        // You get the exception object from the received event
        $exception = $event->getThrowable();
        $message = sprintf(
            'My Error says: %s with code: %s',
            $exception->getMessage(),
            $exception->getCode()
        );

        // Customize your response object to display the exception details
        $response = new Response();
        $response->setContent($message);

        // HttpExceptionInterface is a special type of exception that
        // holds status code and header details
        if ($exception instanceof HttpExceptionInterface) {
            $response->setStatusCode($exception->getStatusCode());
            $response->headers->replace($exception->getHeaders());
        } else {
            $response->setStatusCode(Response::HTTP_INTERNAL_SERVER_ERROR);
        }

        // sends the modified response object to the event
        $event->setResponse($response);
    }
}
```

Рисунок 7 – Пример создания класса для прослушивателя событий

Теперь, когда класс создан, нужно зарегистрировать его как службу и уведомить UDS о том, что он является «прослушивателем» события **kernel.exception**, используя специальный «тег» (Рисунок 8).

```yaml
# config/services.yaml
services:
    App\EventListener\ExceptionListener:
        tags:
            - { name: kernel.event_listener, event: kernel.exception }
```

Рисунок 8 – Пример регистрации класса как службы

2.2.2. Создание подписчика на событие

Другой способ прослушивания событий – через подписчика событий, который представляет собой класс, определяющий один или несколько методов, которые прослушивают одно или различные события. Основное отличие от прослушивателей событий заключается в том, что подписчики всегда знают, какие события они прослушивают.

Если разные методы подписчика события прослушивают одно и то же событие, их порядок определяется параметром **priority**. Это значение является положительным или отрицательным целым числом, которое по умолчанию равно **0**. Чем больше число, тем раньше вызывается метод. Приоритет суммируется для всех прослушивателей и подписчиков, поэтому ваши методы могут быть вызваны до или после методов, определенных в других прослушивателях и подписчиках.

В следующем примере (Рисунок 9). показан подписчик события, который определяет несколько методов, которые прослушивают одно и то же событие **kernel.exception**.

```php
class ExceptionSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        // return the subscribed events, their methods and priorities
        return [
            KernelEvents::EXCEPTION => [
                ['processException', 10],
                ['logException', 0],
                ['notifyException', -10],
            ],
        ];
    }

    public function processException(ExceptionEvent $event)
    {
        // ...
    }

    public function logException(ExceptionEvent $event)
    {
        // ...
    }

    public function notifyException(ExceptionEvent $event)
    {
        // ...
    }
}
```

Рисунок 9 – Пример подписчика событий, определяющего несколько методов

2.3. Рабочий процесс запроса

Рассмотрим, как использовать функции ядра в качестве независимого компонента в любом PHP-приложении. В приложениях UDS все уже настроено и готово к использованию.

Каждое веб-взаимодействие по протоколу HTTP начинается с запроса и заканчивается ответом. Ваша задача как разработчика состоит в том, чтобы создать PHP-код, который считывает информацию о запросе (например, URL), создает и возвращает ответ (например, HTML-страницу или строку JSON) (Рисунок 10).

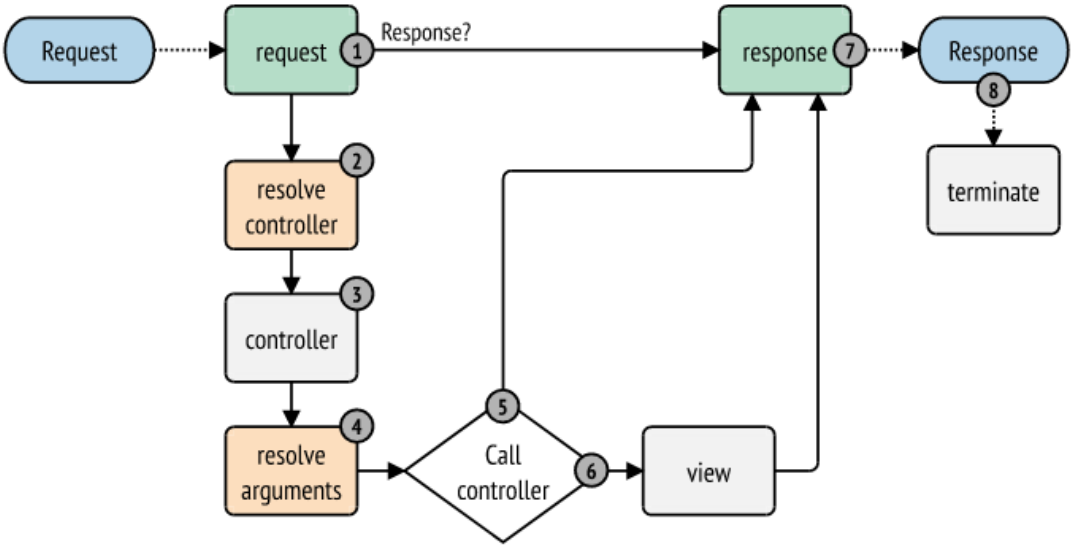

Рисунок 10 – Схема процесса запроса в UDS

1. Пользователь запрашивает ресурс в браузере;
2. Браузер отправляет запрос на сервер;
3. UDS предоставляет приложению объект запроса;
4. Приложение генерирует объект ответа, используя данные объекта запроса;
5. Сервер отправляет ответ обратно в браузер;
6. Браузер отображает ресурс пользователю.

Как правило, создается какой-то фреймворк или система для обработки всех повторяющихся задач (например, маршрутизации, безопасности и т.д.), чтобы разработчик мог создавать каждую страницу приложения.

2.3.3. Объект запроса

Класс **Request** – это объектно-ориентированное представление сообщения HTTP-запроса. С его помощью у вас под рукой есть вся информация о запросе (Рисунок 11).

```php
$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieves $_GET and $_POST variables respectively
$request->query->get('id');
$request->request->get('category', 'default category');

// retrieves $_SERVER variables
$request->server->get('HTTP_HOST');

// retrieves an instance of UploadedFile identified by "attachment"
$request->files->get('attachment');

// retrieves a $_COOKIE value
$request->cookies->get('PHPSESSID');

// retrieves an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content-type');

$request->getMethod();    // e.g. GET, POST, PUT, DELETE or HEAD
$request->getLanguages(); // an array of languages the client accepts
```

Рисунок 11 – Пример использования класса **Request**

2.4. Контроллер

Контроллер – это создаваемая PHP-функция, которая считывает информацию из объекта запроса, создает и возвращает объект ответа. Ответом может быть HTML-страница, JSON, XML, загрузка файла, перенаправление, «Ошибка 404» или что-либо еще. Контроллер выполняет любую произвольную логику, необходимую приложению для отображения содержимого страницы.

2.4.4. Базовый Контроллер

В то время как контроллером может быть любой вызываемый PHP (функция, метод для объекта или замыкание), контроллер обычно является методом внутри класса **controller** (Рисунок 12).

```php
class LuckyController
{
    #[Route('/lucky/number/{max}', name: 'app_lucky_number')]
    public function number(int $max): Response
    {
        $number = random_int(0, $max);

        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

Рисунок 12 – Пример применения контроллера внутри класса **controller**

2.4.5. Сервисный Контейнер

В UDS существует большое количество полезных объектов, например, объект «**Mailer**» может помочь отправлять электронные письма, в то время как другой объект может помочь вам сохранять данные в базе данных. Почти все, что «делает» приложение, на самом деле выполняется одним из этих объектов.

В UDS эти полезные объекты называются сервисами, и каждая служба находится внутри совершенно особого объекта, называемого контейнером сервиса. Контейнер позволяет вам централизовать способ создания объектов. Это облегчает вашу жизнь, способствует созданию надежной архитектуры и выполняется очень быстро.

2.4.6. Получение и использование Сервисов

В тот момент, когда вы запускаете приложение UDS, ваш контейнер уже содержит множество сервисов. Это как инструменты: они ждут, когда вы воспользуетесь ими. В вашем контроллере вы можете «запросить» сервис из контейнера, указав аргумент типа с именем класса сервиса или интерфейса (Рисунок 13).

```php
class ProductController extends AbstractController
{
    #[Route('/products')]
    public function list(LoggerInterface $logger): Response
    {
        $logger->info('Look, I just used a service!');

        // ...
    }
}
```

Рисунок 13 – Пример запроса сервиса из контейнера

2.5. Создание/Настройка Служб в Контейнере

UDS позволяет организовать свой собственный код в сервисы, например, для того, чтобы отобразить пользователям сообщение. В том случае, если поместить этот код в свой контроллер, то он не сможет быть использован повторно. Вместо этого в UDS необходимо создать новый класс (Рисунок 14).

```php
class MessageGenerator
{
    public function getHappyMessage(): string
    {
        $messages = [
            'You did it! You updated the system! Amazing!',
            'That was one of the coolest updates I\'ve seen all day!',
            'Great work! Keep going!',
        ];

        $index = array_rand($messages);

        return $messages[$index];
    }
}
```

Рисунок 14 – Пример создания нового класса

Созданный класс обслуживания можно теперь использовать непосредственно внутри вашего контроллера (Рисунок 15).

```php
class ProductController extends AbstractController
{
    #[Route('/products/new')]
    public function new(MessageGenerator $messageGenerator): Response
    {
        // thanks to the type-hint, the container will instantiate a
        // new MessageGenerator and pass it to you!
        // ...

        $message = $messageGenerator->getHappyMessage();
        $this->addFlash('success', $message);
        // ...
    }
}
```

Рисунок 15 – Пример использования нового класса

2.6. Автоматическая загрузка сервиса в services.yaml

В UDS для нового проекта используется следующая конфигурация сервиса, являющаяся конфигурацией по умолчанию (Рисунок 16).

```
# makes classes in src/ available to be used as services
# this creates a service per class whose id is the fully-qualified class
App\:
    resource: '../src/'
    exclude:
        - '../src/DependencyInjection/'
        - '../src/Entity/'
        - '../src/Kernel.php'

# order is important in this file because service definitions
# always *replace* previous ones; add your own service configuration belo
```

Рисунок 16 – Пример конфигурации сервиса, заданной по умолчанию

2.7. Внедрение Сервисов/Конфигурации в службу

Для того чтобы в UDS получить доступ к службе **logger** из **MessageGenerator** необходимо создать метод **__construct()** с аргументом **$logger**, который имеет тип **LoggerInterface**. Установите это в новом свойстве **$logger** и используйте в дальнейшем (Рисунок 17):

```
class MessageGenerator
{
    public function __construct(
        private LoggerInterface $logger,
    ) {
    }

    public function getHappyMessage(): string
    {
        $this->logger->info('About to find a happy message!');
        // ...
    }
}
```

Рисунок 17 – Пример конфигурации службы

2.7.1. Обработка нескольких Сервисов

Для обработки нескольких Сервисов, например, для того чтобы отправлять электронное письмо администратору сайта каждый раз, когда производится обновление сайта необходимо выполнить создание нового класса (Рисунок 18).

```php
class SiteUpdateManager
{
    public function __construct(
        private MessageGenerator $messageGenerator,
        private MailerInterface $mailer,
    ) {
    }

    public function notifyOfSiteUpdate(): bool
    {
        $happyMessage = $this->messageGenerator->getHappyMessage();

        $email = (new Email())
            ->from('admin@example.com')
            ->to('manager@example.com')
            ->subject('Site update just happened!')
            ->text('Someone just updated the site. We told them: '.$happyMessage);

        $this->mailer->send($email);

        // ...

        return true;
    }
}
```

Рисунок 18 – Пример обработки нескольких Сервисов

ПЕРЕЧЕНЬ ТЕРМИНОВ И СОКРАЩЕНИЙ

PHP – Hypertext Preprocessor;
URL – Uniform Resource Locator;
ORM – Object Relational Mapper;
PSR-4 – Autoloading Standard;
HTTP – HyperText Transfer Protocol;
HTML – HyperText Markup Language;
JSON – JavaScript Object Notation;
XML – eXtensible Markup Language.