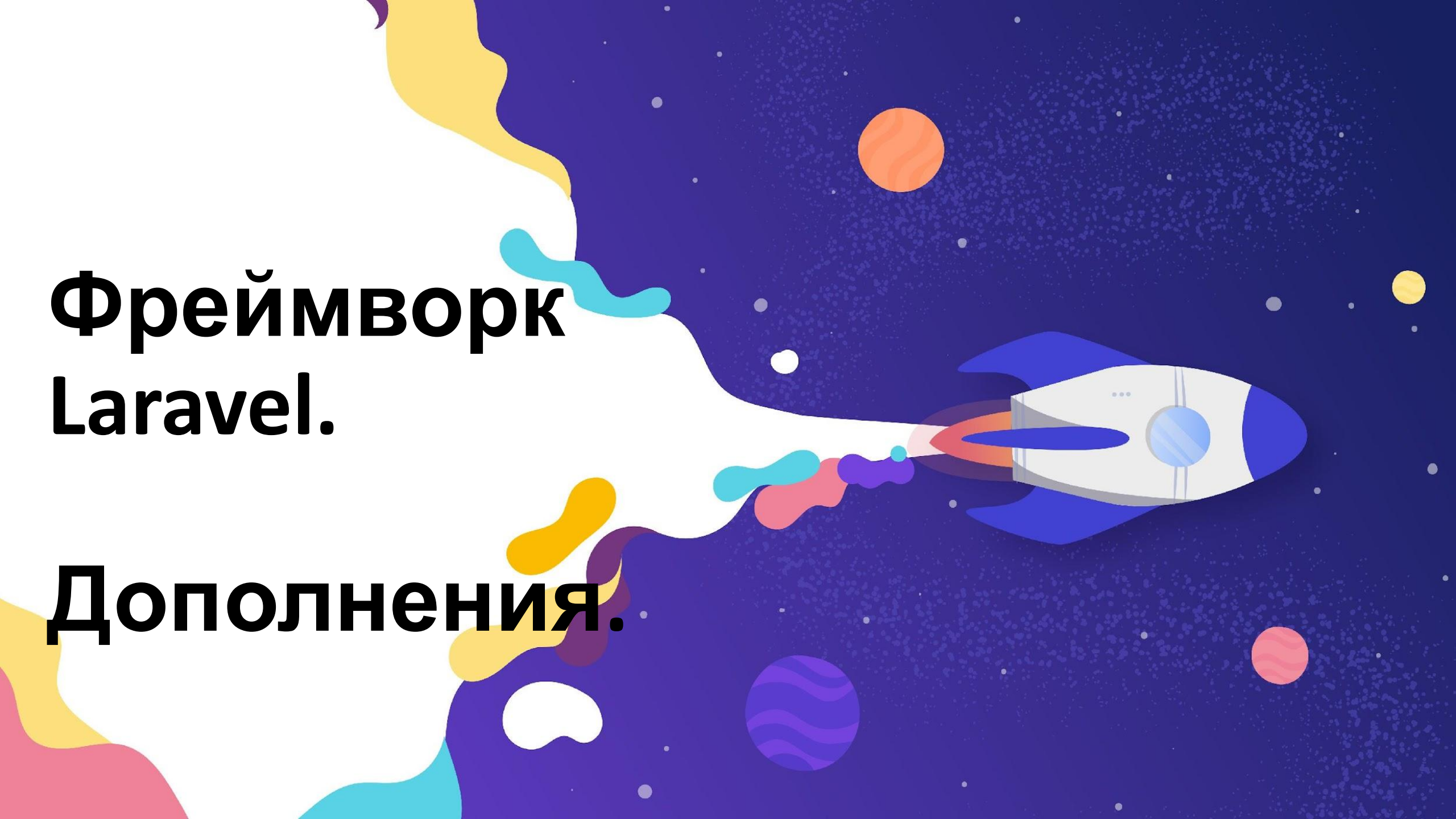


**Фреймворк
Laravel.**

Дополнения.



Фреймворк Laravel. Загрузка файлов

Загрузка файлов - это один из основных элементов многих приложений, которые работают с изображениями, аудио- и видеофайлами, документами и т.д.

В PHP и Laravel существует несколько способов загрузки файлов, которые будут рассмотрены далее.

```
1  <?php
2  if (isset($_FILES['file'])) {
3      $errors = [];
4      $fileName = $_FILES['file']['name'];
5      $fileSize = $_FILES['file']['size'];
6      $fileTmp = $_FILES['file']['tmp_name'];
7      $fileType = $_FILES['file']['type'];
8      $filenamesParts = explode( separator: '.', $_FILES['file']['name']);
9      $file_ext = strtolower(end( &array: $filenamesParts));
10
11     $extensions = ["jpeg", "jpg", "png"];
12
13     if (in_array($file_ext, $extensions) === false) {
14         $errors[] = "Расширение файла запрещено. Пожалуйста, выберите JPEG или PNG файл.";
15     }
16
17     if ($fileSize > 2097152) {
18         $errors[] = 'Размер файла должен быть не более 2 MB';
19     }
20
21     if (empty($errors)) {
22         move_uploaded_file($fileTmp, "uploads/" . $fileName);
23         echo "Файл загружен успешно!";
24     } else {
25         print_r($errors);
26     }
27 }
```

Фреймворк Laravel. Загрузка файлов

В этом примере используется класс `Illuminate\Http\Request` для получения файла из запроса.

Затем происходит проверка файла на тип и размер с помощью метода `validate()`. Если проверка прошла успешно, то файл перемещается в директорию `public/uploads`.

```
7 public function store(Request $request): View
8 {
9     $request->validate([
10         'file' => 'required|mimes:jpeg,png|max:2048',
11     ]);
12
13     $filename = time().'.'.$request->file->extension();
14
15     $request->file->move(public_path('uploads'), $filename);
16
17     return view('file.uploaded', ['filename' => $filename]);
18 }
```

Важно: необходимо создать директорию `public/uploads` перед использованием метода `move()`, иначе Laravel выдаст исключение.

Фреймворк Laravel. Кэширование

Кэширование - это механизм, который позволяет сохранять результаты вычислений или запросов к базе данных в памяти сервера, чтобы уменьшить время ответа и ускорить работу приложения. Laravel предоставляет несколько способов кэширования, включая файловое, базы данных, Redis, и другие.

Рассмотрим пример использования кэширования в Laravel. Допустим, есть страница с отображением списка товаров, которые выбираются из базы данных. Запрос к базе данных может быть достаточно медленным, особенно если в базе данных находится много записей. Чтобы ускорить работу страницы, можно воспользоваться механизмом кэширования и записать результат запроса в кэш на некоторое время.

Фреймворк Laravel. Кэширование

Для начала, необходимо настроить кэширование в Laravel. Для этого в файле конфигурации `config/cache.php` необходимо выбрать драйвер кэширования. Для примера мы выберем драйвер "file", который сохраняет кэшированные данные в файловой системе.

Для кэширования результатов запроса к базе данных в Laravel можно использовать фасад `Cache`.

```
use Illuminate\Support\Facades\Cache;
use App\Models\Product;

$products = Cache::remember( key: 'products', ttl: 60, function () {
    return Product::all();
});

return view( view: 'products.index', ['products' => $products]);
```

Фреймворк Laravel. Кэширование

В этом примере используется метод `Cache::remember`, который сохраняет результат выполнения функции в кэше на заданный промежуток времени (в данном случае 60 секунд). Если данные уже были закэшированы, метод вернет их без выполнения функции.

Таким образом, при первом запросе к странице выполнится запрос к базе данных и результаты попадут в кэш на 60 секунд.

При последующих запросах кэшированные данные будут получены намного быстрее, что ускорит работу приложения.

Стоит отметить, что работать с кэшем необходимо аккуратно: кэшировать только те данные, которые не будут изменяться в течение времени жизни кэша.

В противном случае необходимо воспользоваться механизмом инвалидации кэша.

Фреймворк Laravel. Кэширование

Инвалидация кэша - это процесс удаления сохраненных в кэше данных, чтобы обеспечить их актуальность и соответствие текущему состоянию приложения. В Laravel есть несколько способов инвалидации кэша, включая прямую инвалидацию, тэгирование, события и другие.

Для начала рассмотрим пример использования прямой инвалидации в Laravel. Предположим, есть страница с отображением информации о продукте, которая находится в кэше, чтобы ускорить ее загрузку. Однако, если информация о продукте изменится, закэшированные данные станут недействительными и потребуется их инвалидация.

В Laravel существует метод `Cache::forget`, который используется для удаления кэшированных данных.

Фреймворк Laravel. Кэширование

```
use Illuminate\Support\Facades\Cache;
use App\Models\Product;

$product = Product::find(1);
Cache::forget('product_' . $product->id);

// Обновление информации о продукте
$product->name = 'Новое имя продукта';
$product->save();
```

В этом примере используется метод `Cache::forget` для удаления кэшированных данных о продукте с указанным идентификатором.

После этого информация о продукте обновляется и эта информация будет использована при следующем запросе.

Фреймворк Laravel. Кэширование

```
use Illuminate\Support\Facades\Cache;
use App\Models\Product;

$product = Product::find(1);
Cache::tags(['products', 'product_' . $product->id])->forget();

// Обновление информации о продукте
$product->name = 'Новое имя продукта';
$product->save();
```

Кроме того, в Laravel можно использовать тэгирование кэша, чтобы связать кэшированные данные с определенным тэгом.

Это позволяет инвалидировать несколько элементов кэша одновременно, используя только один тэг. Для этого можно использовать методы `Cache::tags` и `Cache::flush`

В этом примере используется метод `Cache::tags` для связывания элементов кэша с тэгом "products" и "product_{id}".

После этого используется метод `Cache::forget` для удаления всех элементов кэша, связанных с этими тэгами.

При следующем запросе кэш будет создан заново с обновленной информацией.

Фреймворк Laravel. Брокеры сообщений

Брокеры сообщений - это инструмент, который позволяет отправлять сообщения между разными компонентами приложения, обеспечивая связь между ними.

В Laravel доступны несколько встроенных брокеров сообщений, таких как Redis, RabbitMQ, Amazon SQS и другие. Каждый брокер сообщений имеет свои особенности и преимущества, поэтому выбор брокера зависит от конкретных требований проекта.

Рассмотрим пример использования брокера сообщений в Laravel. Предположим, есть система уведомлений, которая должна отправлять уведомления пользователям о различных событиях, таких как новые сообщения, новые заказы и другие. Для отправки уведомлений можно использовать брокер сообщений Redis.

Фреймворк Laravel. Брокеры сообщений

Для начала, необходимо установить драйвер Redis для брокера сообщений. Для этого мы можем использовать менеджер очередей Laravel. Для настройки менеджера очередей необходимо отредактировать файл `.env` и добавить следующие строки:

```
QUEUE_CONNECTION=redis
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

Фреймворк Laravel. Брокеры сообщений

После этого необходимо создать очередь уведомлений и добавить в нее задачи на отправку уведомлений. Для этого используется метод `dispatch` и указывается класс задачи, которая будет выполнена при отправке уведомления:

```
use App\Jobs\SendNotification;  
use Illuminate\Support\Facades\Queue;  
  
$user = User::find(1);  
Queue::push(new SendNotification($user, 'У вас новое сообщение'));
```

В этом примере используется метод `Queue::push` для добавления задачи на отправку уведомления в очередь.

Класс задачи `SendNotification` должен реализовывать интерфейс `ShouldQueue` и содержать логику отправки уведомления.

Фреймворк Laravel. Брокеры сообщений

После добавления задачи в очередь, необходимо запустить процесс обработки очереди. Для этого используется команда `queue:work`:

```
php artisan queue:work
```

Эта команда запустит процесс обработки очереди, который будет выполнен на протяжении всего времени работы приложения.

Как только задача будет добавлена в очередь, она будет обработана и уведомление будет отправлено.

Фреймворк Laravel. Artisan console

Artisan Console - инструмент командной строки, который позволяет выполнять различные задачи в Laravel-приложении.

Он предоставляет множество команд, которые могут помочь управлять приложением, от создания контроллеров до миграций баз данных. Большинство команд Artisan можно запустить с помощью консоли командной строки, что делает их использование быстрым и удобным.

Предположим, есть приложение, которое использует базу данных MySQL и необходимо создать новую миграцию для добавления новой таблицы в базу данных. Для создания новой миграции можно использовать команду `make:migration`:

```
php artisan make:migration create_products_table
```

Фреймворк Laravel. Artisan console

Эта команда создаст новую миграцию в папке `database/migrations`, которую мы можем отредактировать, чтобы добавить необходимую логику.

Кроме того, мы можем использовать Artisan Console для выполнения тестов и отладки приложения. Для выполнения тестов мы можем использовать команду `test`, которая запустит все тесты в приложении:

```
php artisan test
```

Для отладки приложения можно воспользоваться командой `tinker`, которая открывает интерактивную оболочку, где можно выполнять PHP-код и проверять работу приложения:

```
php artisan tinker
```

Фреймворк Laravel. Artisan console

Также Artisan Console позволяет создавать новые контроллеры, маршруты и другие компоненты приложения, используя команды

- `make:controller`
- `make:model`
- `make:request`

и другие.

Фреймворк Laravel. Artisan console

Laravel обеспечивает возможность создания собственной консольной команды. Для этого необходимо воспользоваться командой `make:command`, указав имя команды:

```
php artisan make:command MyCommand
```

После этого Laravel создаст новый класс `MyCommand` в папке `app/Console/Commands`, который можно отредактировать, чтобы добавить необходимую логику.

Команда должна содержать метод `handle`, который будет выполнять основную логику команды. В этом методе можно использовать различные методы и сервисы Laravel, чтобы выполнить нужные действия.

Фреймворк Laravel. Artisan console

Кроме того, мы можем добавить аргументы и опции к нашей команде, чтобы сделать ее более гибкой. Например, мы можем добавить опцию `--count`, которая будет указывать количество элементов, которые нужно обработать:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

no usages
class MyCommand extends Command
{
    protected $signature = 'mycommand [--count=10 : Количество элементов]';

    public function handle(): void
    {
        $count = $this->option( key: 'count');

        for ($i = 1; $i <= $count; $i++) {
            // Обработка элементов
        }
    }
}
```

В этом примере добавлена опция `--count`, которая по умолчанию равна 10.

Затем значение этой опции используется в методе `handle` для обработки элементов.

Фреймворк Laravel. Task Scheduling

Task Scheduling - это механизм, который позволяет выполнять задачи по расписанию в Laravel.

С помощью Task Scheduling можно запускать команды Artisan Console автоматически в заданное время или с определенной периодичностью.

Система Linux также предусматривает механизм для таких целей.

Crontab - это утилита на серверах Linux, которая позволяет запускать задания по расписанию. Она работает в фоновом режиме и выполняет задания в указанное время.

Для создания новой задачи в Crontab необходимо выполнить команду `crontab -e`. Эта команда откроет файл Crontab в текстовом редакторе, где можно добавить новую задачу.

Фреймворк Laravel. Task Scheduling

Формат строки задачи в Crontab выглядит следующим образом:

```
* * * * * command
```

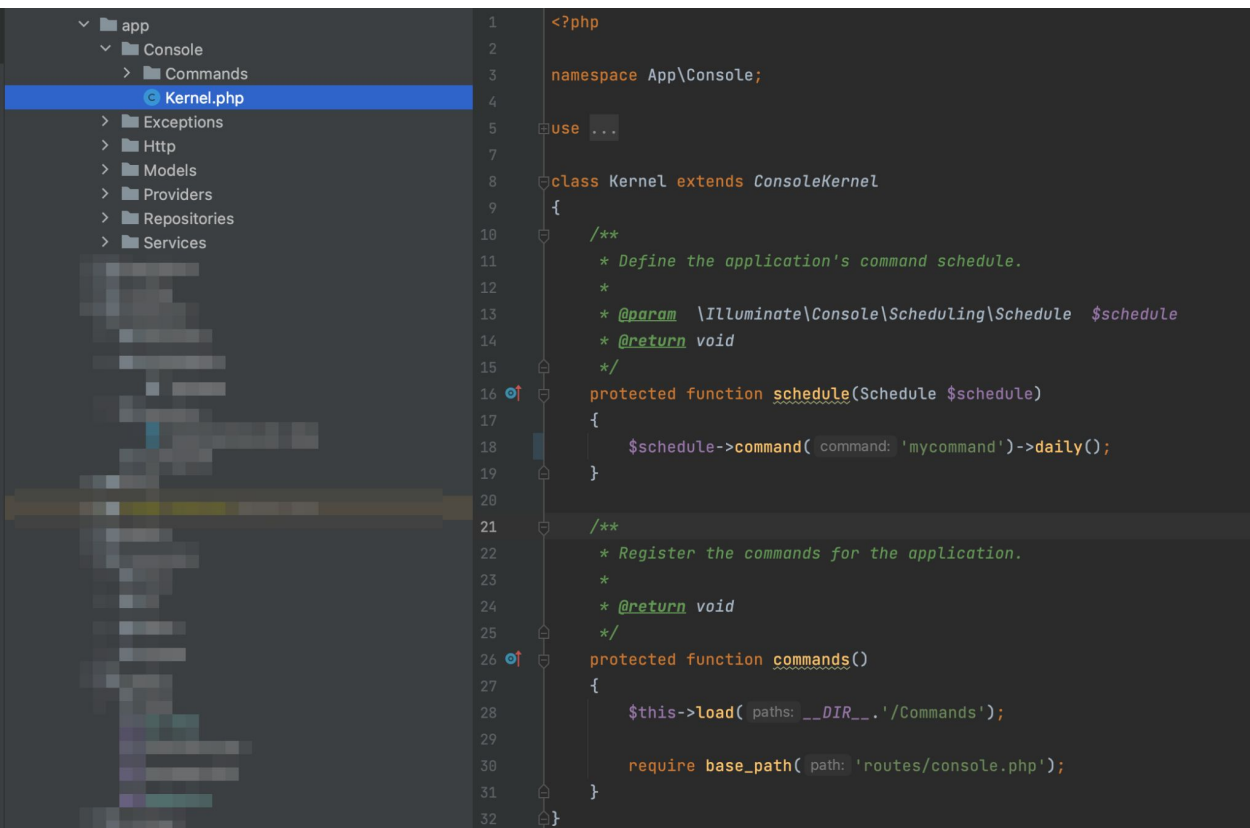
В этом формате первые пять значений представляют собой время запуска задачи, а последний параметр - команда, которую нужно выполнить. Каждый параметр разделен пробелом.

Параметры времени могут принимать следующие значения:

- * - любое значение
- */n - каждые n единиц времени (например, */5 для запуска каждые 5 минут)
- n - конкретное значение времени (например, 30 для запуска в 30 минут каждого часа)

Фреймворк Laravel. Task Scheduling

Чтобы настроить выполнение заданий Artisan по расписанию можно воспользоваться механизмом crontab в самой ОС, либо встроенным механизмом Task Scheduling в Laravel.



```
1 <?php
2
3 namespace App\Console;
4
5 use ...
6
7
8 class Kernel extends ConsoleKernel
9 {
10     /**
11      * Define the application's command schedule.
12      *
13      * @param \Illuminate\Console\Scheduling\Schedule $schedule
14      * @return void
15      */
16     protected function schedule(Schedule $schedule)
17     {
18         $schedule->command('mycommand')->daily();
19     }
20
21     /**
22      * Register the commands for the application.
23      *
24      * @return void
25      */
26     protected function commands()
27     {
28         $this->load(paths: __DIR__.'/Commands');
29
30         require base_path('routes/console.php');
31     }
32 }
```

Для этого необходимо открыть файл `app/Console/Kernel.php` и зарегистрировать задачи в методе `schedule`.

В этом примере команда `mycommand` сконфигурирована для выполнения ежедневно.

Также доступны и другие конфигурации: `everyMinute`, `everyFiveMinutes`, `hourly`, `weekly` и другие

Фреймворк Laravel. Middleware

Middleware - это промежуточное звено между запросом и ответом в Laravel. Оно позволяет обрабатывать запросы перед их выполнением и после выполнения, например, для аутентификации, проверки прав доступа, логирования и многое другое.

Для создания Middleware в Laravel мы можем использовать команду `make:middleware`. Например, чтобы создать Middleware для аутентификации, необходимо выполнить следующую команду:

```
php artisan make:middleware Authenticate
```

Эта команда создаст новый класс `Authenticate` в папке `app/Http/Middleware`, который можно редактировать для добавления нужной логики.

Фреймворк Laravel. Middleware

Middleware в Laravel 9 выполняются в порядке их регистрации, поэтому важно учитывать порядок регистрации, особенно если Middleware зависят друг от друга. Регистрацию Middleware можно выполнить в файле `app/Http/Kernel.php` в свойстве `$middleware`.

Для регистрации Middleware перед выполнением запроса, можно использовать свойство `$middleware`:

```
class UserController extends Controller
{
    protected $middleware = [
        Authenticate::class,
        // Другие Middleware
    ];
}
```


Фреймворк Laravel. Middleware

Middleware в Laravel 9 выполняются в порядке их регистрации, поэтому важно учитывать порядок регистрации, особенно если Middleware зависят друг от друга. Регистрацию Middleware можно выполнить в файле `app/Http/Kernel.php` в свойстве `$middleware`.

Для регистрации Middleware перед выполнением запроса, можно использовать свойство `$middleware`:

```
class UserController extends Controller
{
    protected $middleware = [
        Authenticate::class,
        // Другие Middleware
    ];
}
```

В этом примере Middleware `Authenticate` будет выполнено перед выполнением запроса.

Фреймворк Laravel. Middleware

Для регистрации Middleware после выполнения запроса, используется свойство `$middlewareAfter`:

```
class UserController extends Controller
{
    protected $middleware = [
        Authenticate::class,
        // Другие Middleware
    ];

    no usages
    protected array $middlewareAfter = [
        LogResponse::class,
        // Другие Middleware
    ];
}
```

В этом примере Middleware `LogResponse` будет выполнено после выполнения запроса и перед выдачей ответа.

Фреймворк Laravel. Middleware

Middleware в Laravel 9 могут использоваться как глобально, так и локально для определенных маршрутов или контроллеров. Для локального использования Middleware в маршрутах или контроллерах можно использовать метод `middleware`:

```
api.php x
1 <?php
2
3 use Illuminate\Support\Facades\Route;
4
5 Route::get( uri: 'users', [\App\Http\Controllers\UserController::class, 'getUsersForApi'])->middleware( middleware: 'Authenticate');
```

Этот пример показывает, как Middleware `Authenticate` будет применяться только к маршруту `/users`.

Фреймворк Laravel. Routing

Маршрутизация - это процесс определения того, какие действия должны быть выполнены при обработке запроса в Laravel. В Laravel маршрутизация осуществляется с помощью файлов, находящихся в папке `routes`, в которой содержатся файлы с маршрутами для разных режимов работы приложения.

Например, для web части, необходимо использовать файл `routes/web.php`, а для api - `routes/api.php`. Важно: у api - маршрутов будет префикс `api`. Например, при задании маршрута `/users`, этот маршрут будет доступен, как `/api/users`.

Для создания маршрута используется функция `Route::METHOD()`, где `METHOD` - это метод HTTP-запроса (GET, POST, PUT, DELETE и т.д.). Например, чтобы создать маршрут для GET-запроса, можно использовать следующий код:

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

Фреймворк Laravel. Routing

Маршруты в Laravel могут иметь параметры, которые могут быть переданы в запросе. Например, для создания маршрута, который принимает параметр `id`, напомним следующий код:

```
Route::get( uri: '/users/{id}', function ($id) {  
    return 'User with ID ' . $id;  
});
```

В этом примере создается маршрут для URL `/users/{id}`, который принимает параметр `id` и возвращает строку "User with ID {id}".

Фреймворк Laravel. Routing

Маршруты в Laravel также могут быть определены с помощью контроллеров, что делает код более структурированным и читаемым. Например, можно определить маршрут с помощью контроллера `UserController`:

```
Route::get( uri: 'users', [\App\Http\Controllers\UserController::class, 'getUsersForApi']);
```

В этом примере определяется маршрут для URL `/users` с помощью метода `getUsersForApi` контроллера `UserController`.

Кроме того, в Laravel можно создавать группы маршрутов, которые позволяют группировать маршруты с общим префиксом или Middleware. Например, для создания группы маршрутов с Middleware для аутентификации подойдет следующий код:

```
Route::middleware(['auth'])->group(function () {  
    Route::get( uri: '/dashboard', [DashboardController::class, 'index']);  
    Route::get( uri: '/profile', [ProfileController::class, 'index']);  
});
```

В этом примере создана группа маршрутов, которые требуют аутентификации с помощью Middleware `auth`.

Внутри группы определены маршруты `/dashboard` и `/profile`, которые будут доступны только после успешной аутентификации.