

LARAVEL

СОДЕРЖАНИЕ

1. INSTALLATION	5
LARAVEL & DOCKER	6
НАЧАЛО РАБОТЫ В WINDOWS	6
НАЧАЛЬНАЯ КОНФИГУРАЦИЯ	7
КОНФИГУРАЦИЯ НА ОСНОВЕ СРЕДЫ	7
БАЗЫ ДАННЫХ И МИГРАЦИЯ	8
КОНФИГУРАЦИЯ КАТАЛОГА	9
ПОДДЕРЖКА IDE	9
УСТАНОВКА НА ВИНДОВС ЧЕРЕЗ OPENSERVR.....	9
ЭТАПЫ УСТАНОВКИ	9
2. CONFIGURATION	13
КОНФИГУРАЦИЯ СРЕДЫ.....	14
ТИПЫ ПЕРЕМЕННЫХ.....	15
ШИФРОВАНИЕ ФАЙЛОВ СРЕДЫ	17
РАСШИФРОВКА.....	17
ДОСТУП К ЗНАЧЕНИЯМ КОНФИГУРАЦИИ	19
КЭШИРОВАНИЕ КОНФИГУРАЦИИ	19
3. DIRECTORY STRUCTURE	20
КОРНЕВОЙ КАТАЛОГ	20
• Каталог приложений	20
• Каталог начальной загрузки.....	21
• Каталог конфигурации	21
• Каталог базы данных	21
• Публичный каталог	21
• Каталог ресурсов	21
• Каталог маршрутов	21
• Каталог хранений.....	22
• Каталог тестов	22
• Каталог поставщиков.....	22
• Каталог вещаний.....	22
• Каталог консоли	23
• Каталог событий	23
• Каталог исключений	23

• HTTP каталог.....	23
• Каталог вакансий.....	23
• Каталог слушателей	23
• Почтовый каталог.....	24
• Каталог моделей	24
• Каталог уведомлений	24
• Каталог политик	24
• Каталог правил.....	24
4. FRONTEND	25
Введение.....	25
Использование PHP	25
PHP и Blade.....	25
Рис.19 пример.....	26
Ожидания.....	26
Livewire.....	26
Использование Vue/React	28
Inertia.....	28
Серверный рендеринг	30
Объединение активов.....	30
5. СТАРТОВЫЕ НАБОРЫ.....	31
Laravel Breeze	31
Установка	31
Сначала вам следует создать новое приложение Laravel, настроить базу данных и выполнить миграцию базы данных . После создания нового приложения Laravel вы можете установить Laravel Breeze с помощью Composer:.....	
Breeze и Next.js/API.....	32
Laravel Jetstream.....	32
6. РАЗВЕРТЫВАНИЕ.....	33
Конфигурация сервера.....	34
Ngnx.....	34
Оптимизация	35
Оптимизация автозагрузчика.....	35
Конфигурация кэширования	35
Кэширование событий	35
Кэширование представлений.....	36
Режим отладки.....	36
Простое развертывание с помощью Forge / Vapor	36
Laravel Forge	36

Laravel Vapor.....	36
7. Жизненный цикл запроса.....	37
Первые шаги.....	37
HTTP/консольные ядра.....	37
Поставщики услуг.....	38
Маршрутизация.....	38
Заканчивать.....	39
Сосредоточьтесь на поставщиках услуг.....	39
8. СЕРВИСНЫЙ КОНТЕЙНЕР.....	40
Разрешение нулевой конфигурации.....	41
Когда использовать контейнер.....	42
Связывание.....	42
Основные привязки.....	42
Простые привязки.....	42
Привязка синглтона.....	44
Привязка синглов с ограниченной областью действия.....	44
Экземпляр привязки.....	45
Привязка интерфейсов к реализациям.....	45
Контекстная привязка.....	46
Связывание примитивов.....	47
Привязка типизированных вариативов.....	48
Зависимости вариативных тегов.....	49
Тегирование.....	50
Расширение привязок.....	50
Разрешение.....	51
Метод <code>make</code>	51
Автоматическая инъекция.....	52
Вызов метода и внедрение.....	53
Контейнерные события.....	55
PSR-11.....	55
9. ПОСТАВЩИКИ УСЛУГ.....	56
Поставщики услуг письма.....	56
Метод регистрации.....	56
Свойства <code>bindings</code> и <code>singletons</code>	57
Метод загрузки.....	58
Внедрение зависимостей метода загрузки.....	59
Регистрация провайдеров.....	60

Отложенные поставщики	60
10. FACADE	62
Вспомогательные функции	62
Когда использовать фасады	63
Фасады против. Внедрение зависимости.....	63
Фасады против. Вспомогательные функции	64
Как работают фасады.....	65
Фасады реального времени	67
Справочник классов фасадов	70
11.....	73
ПРОДОЛЖЕНИЕ ЕСТЬ, НО ОНО НЕ НАПИСАНО ТУТ	

LARAVEL

1. INSTALLATION

Laravel - фреймворк обеспечивает структуру и отправную точку для создания вашего приложения, внедряет зависимости, выразительный уровень абстракции базы данных, очереди и запланированные задания, модульное и интеграционное тестирование и многое другое.

Laravel невероятно масштабируем. Благодаря удобному масштабированию PHP и встроенной поддержке Laravel быстрых систем распределенного кэширования, таких как Redis, горизонтальное масштабирование с помощью Laravel выполняется очень просто. Фактически, приложения Laravel легко масштабируются для обработки сотен миллионов запросов в месяц.

Первый проект

Прежде чем создавать свой первый проект Laravel, вы должны убедиться, что на вашем локальном компьютере установлены PHP и Composer . Если вы разрабатываете для macOS, PHP и Composer можно установить за считанные минуты с помощью Laravel Herd . Кроме того, мы рекомендуем установить Node и NPM .

После установки PHP и Composer вы можете создать новый проект Laravel с помощью create-project команды Composer:

```
composer create-project laravel/laravel example-app
```

Рис.1. Создание

Или вы можете создавать новые проекты Laravel, глобально установив установщик Laravel через Composer. Или, если вы установили PHP и Composer через Laravel Herd , вам уже доступен установщик Laravel:

```
composer global require laravel/installer

laravel new example-app
```

Рис.2. установка

После создания проекта запустите локальный сервер разработки Laravel с помощью `serve` команды Artisan CLI Laravel:



```
cd example-app

php artisan serve
```

Рис.3. запуск

После запуска сервера разработки Artisan ваше приложение станет доступно в веб-браузере по адресу `http://localhost:8000`. Далее вы готовы сделать следующие шаги в экосистеме Laravel . Конечно, вы также можете настроить базу данных .

LARAVEL & DOCKER

Docker — это инструмент для запуска приложений и служб в небольших, легких «контейнерах», которые не мешают установленному программному обеспечению или конфигурации вашего локального компьютера. Это означает, что вам не нужно беспокоиться о настройке сложных инструментов разработки, таких как веб-серверы и базы данных, на вашем локальном компьютере. Для начала вам нужно всего лишь установить Docker Desktop .

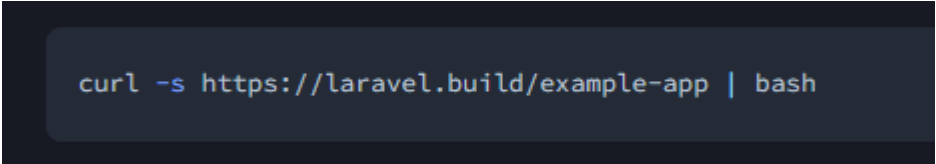
Laravel Sail — это легкий интерфейс командной строки для взаимодействия с конфигурацией Docker Laravel по умолчанию. Sail обеспечивает отличную отправную точку для создания приложения Laravel с использованием PHP, MySQL и Redis без предварительного опыта работы с Docker.

НАЧАЛО РАБОТЫ В WINDOWS

Прежде чем мы создадим новое приложение Laravel на вашем компьютере с Windows, обязательно установите Docker Desktop . Далее вам следует убедиться, что подсистема Windows для Linux 2 (WSL2) установлена и включена. WSL позволяет запускать двоичные исполняемые файлы Linux в Windows 10. Информацию о том, как установить и включить WSL2, можно найти в документации среды разработки Microsoft .

После установки и включения WSL2 необходимо убедиться, что Docker Desktop настроен на использование серверной части WSL2

Далее вы готовы создать свой первый проект Laravel. Запустите терминал Windows и начните новый сеанс терминала для вашей операционной системы WSL2 Linux. Далее вы можете использовать простую команду терминала для создания нового проекта Laravel. Например, чтобы создать новое приложение Laravel в каталоге с именем «example-app», вы можете запустить в своем терминале следующую команду:



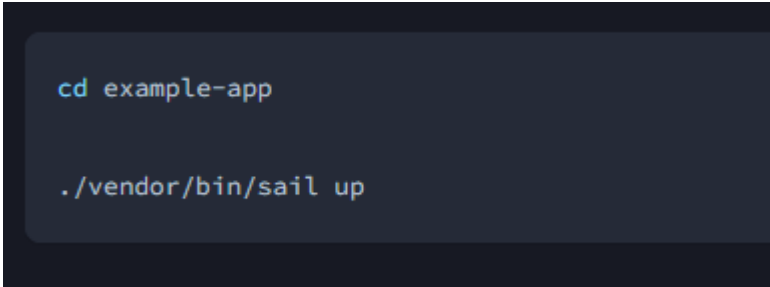
```
curl -s https://laravel.build/example-app | bash
```

Рис.4. проект laravel

Конечно, вы можете изменить «example-app» в этом URL-адресе на что угодно — просто убедитесь, что имя приложения содержит только буквенно-цифровые символы, тире и подчеркивания. Каталог приложения Laravel будет создан в каталоге, из которого вы выполняете команду.

Установка Sail может занять несколько минут, пока контейнеры приложений Sail собираются на вашем локальном компьютере.

После создания проекта вы можете перейти в каталог приложения и запустить Laravel Sail. Laravel Sail предоставляет простой интерфейс командной строки для взаимодействия с конфигурацией Docker Laravel по умолчанию:



```
cd example-app  
  
./vendor/bin/sail up
```

Рис.5. запуск docker контейнеров

После запуска Docker-контейнеров приложения вы можете получить доступ к приложению в веб-браузере по адресу: <http://localhost>

НАЧАЛЬНАЯ КОНФИГУРАЦИЯ

Все файлы конфигурации платформы Laravel хранятся в этом config каталоге. Каждая опция документирована, поэтому не стесняйтесь просматривать файлы и знакомиться с доступными вам опциями.

Laravel практически не требует дополнительной настройки «из коробки». Вы можете начать разработку! Однако вы можете просмотреть config/app.php файл и его документацию. Он содержит несколько опций, таких как timezone и locale которые вы можете изменить в соответствии с вашим приложением.

КОНФИГУРАЦИЯ НА ОСНОВЕ СРЕДЫ

Поскольку многие значения параметров конфигурации Laravel могут различаться в зависимости от того, работает ли ваше приложение на локальном компьютере или на рабочем веб-сервере, многие важные значения

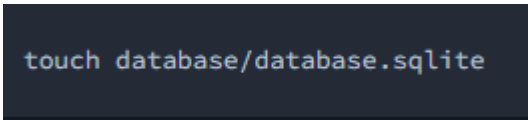
конфигурации определяются с использованием файла, .env который существует в корне вашего приложения.

Ваш .env файл не должен быть передан в систему контроля версий вашего приложения, поскольку каждому разработчику/серверу, использующему ваше приложение, может потребоваться другая конфигурация среды. Кроме того, это может представлять угрозу безопасности в случае, если злоумышленник получит доступ к вашему хранилищу системы контроля версий, поскольку любые конфиденциальные учетные данные будут раскрыты.

БАЗЫ ДАННЫХ И МИГРАЦИЯ

Теперь, когда вы создали приложение Laravel, вы, вероятно, захотите сохранить некоторые данные в базе данных. По умолчанию в файле конфигурации вашего приложения .envуказано, что Laravel будет взаимодействовать с базой данных MySQL и получит доступ к базе данных по адресу 127.0.0.1. Если вы разрабатываете для macOS и вам необходимо установить MySQL, Postgres или Redis локально, вам может быть удобно использовать DBngin .

Если вы не хотите устанавливать MySQL или Postgres на свой локальный компьютер, вы всегда можете использовать базу данных SQLite . SQLite — это небольшой, быстрый и автономный движок базы данных. Для начала создайте базу данных SQLite, создав пустой файл SQLite. Обычно этот файл находится в database каталоге вашего приложения Laravel:



```
touch database/database.sqlite
```

Рис.6. пример кода

Затем обновите .env файл конфигурации, чтобы использовать драйвер базы данных Laravel sqlite. Вы можете удалить другие параметры конфигурации базы данных:


```
DB_CONNECTION=sqlite
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

Рис.7. пример

После настройки базы данных SQLite вы можете запустить миграцию базы данных вашего приложения , в результате которой будут созданы таблицы базы данных вашего приложения:

```
php artisan migrate
```

Рис.8. пример

КОНФИГУРАЦИЯ КАТАЛОГА

Laravel всегда должен обслуживаться из корня «веб-каталога», настроенного для вашего веб-сервера. Вам не следует пытаться обслуживать приложение Laravel из подкаталога «веб-каталога». Попытка сделать это может привести к раскрытию конфиденциальных файлов, присутствующих в вашем приложении.

ПОДДЕРЖКА IDE

При разработке приложений Laravel вы можете использовать любой редактор кода по вашему желанию; однако PhpStorm предлагает обширную поддержку Laravel и его экосистемы, включая Laravel Pint .

Кроме того, поддерживаемый сообществом плагин Laravel Idea PhpStorm предлагает множество полезных дополнений IDE, включая генерацию кода, завершение синтаксиса Eloquent, завершение правил проверки и многое другое.

УСТАНОВКА НА ВИНДОВС ЧЕРЕЗ OPENSERVER

ЭТАПЫ УСТАНОВКИ

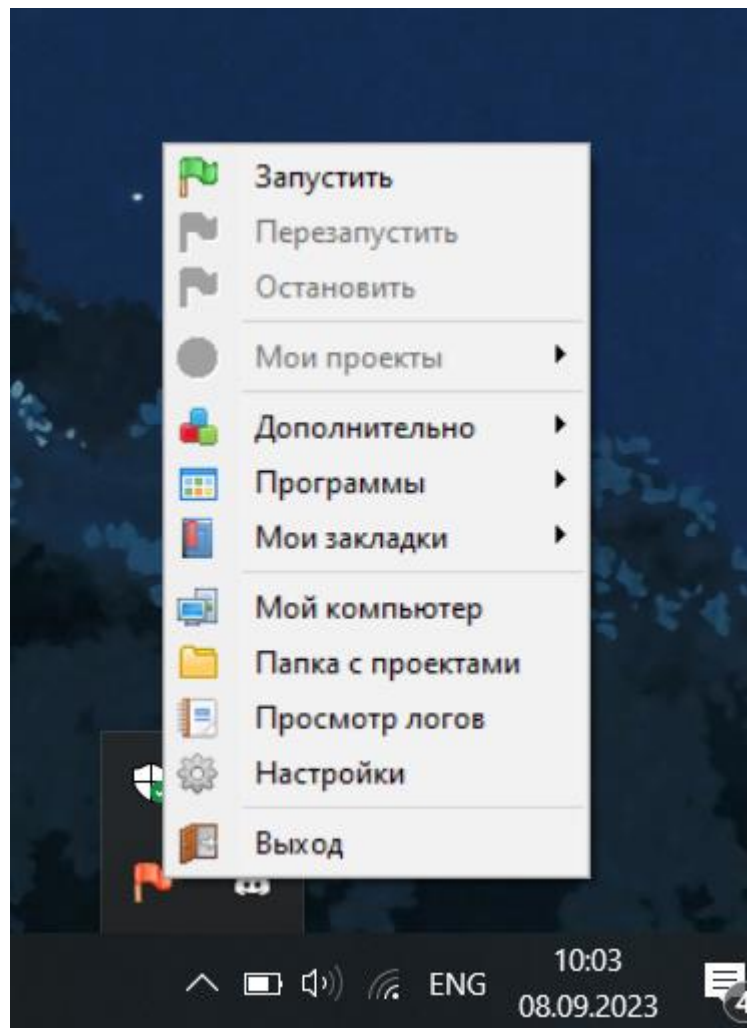


Рис.1 запуск

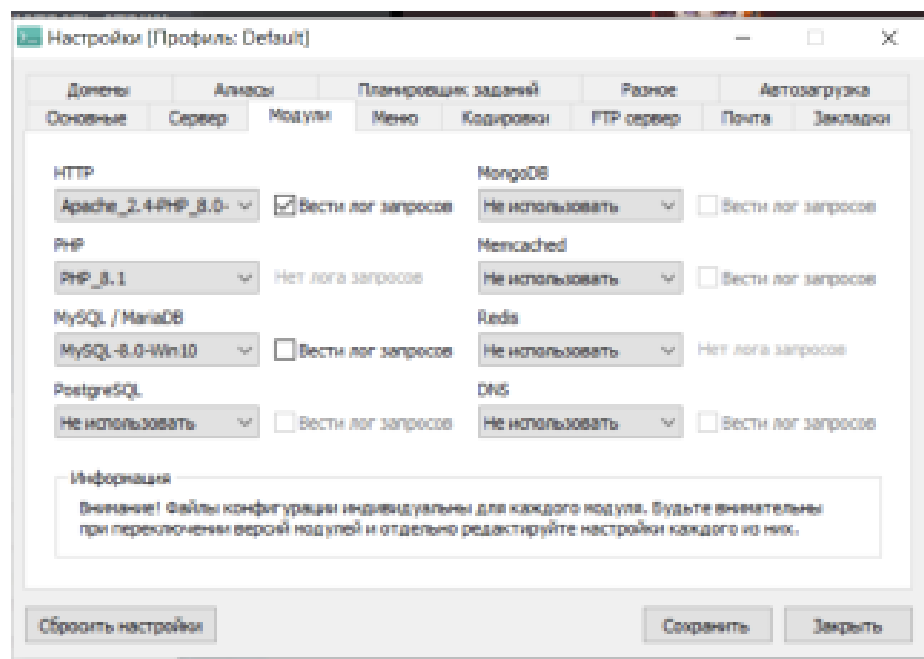


Рис.2 проверка настроек

Вводим команду в консоли openserver

composer create-project laravel/laravel example-app

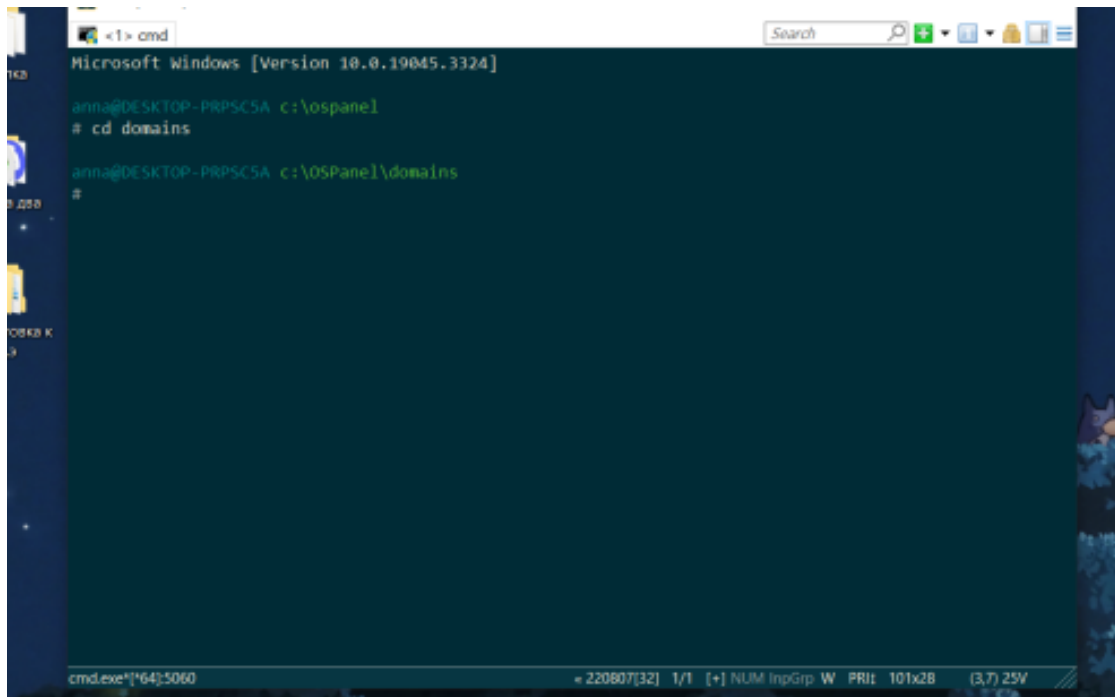


Рис.3. консоль и команда

Вводим команду

Cd example-app

php artisan about

```
cmd (Admin)
> @php artisan key:generate --ansi
INFO: Application key set successfully.

anna@DESKTOP-PRPSC5A c:\OSPanel\domains
# cd example-app

anna@DESKTOP-PRPSC5A c:\OSPanel\domains\example-app
# php artisan about

Environment .....
Application Name ..... Laravel
Laravel Version ..... 10.22.0
PHP Version ..... 8.1.9
Composer Version ..... 2.4-dev+28b3e3e79c5dd62d65b0eafe3d968a31a05b1c31
Environment ..... local
Debug Mode ..... ENABLED
URL ..... localhost
Maintenance Mode ..... OFF

Cache .....
Config ..... NOT CACHED
Events ..... NOT CACHED
Routes ..... NOT CACHED
Views ..... NOT CACHED

Drivers .....
```

Рис.4. процесс в консоли

```
cmd (Admin)
anna@DESKTOP-PRPSC5A c:\ospanel
# cd domains

anna@DESKTOP-PRPSC5A c:\OSPanel\domains
# composer create-project laravel/laravel example-app
Warning: This development build of Composer is over 60 days old. It is recommended to update it by running
"ci:\ospanel\modules\php\PHP 8.1\..\..\userdata\composer\composer.phar self-update" to get the latest version.

Creating a "laravel/laravel" project at "./example-app"
Info from https://repo.packagist.org: #standwithukraine
Installing laravel/laravel (v10.2.6)
- Downloading laravel/laravel (v10.2.6)
- Installing laravel/laravel (v10.2.6): Extracting archive
Created project in C:\OSPanel\domains\example-app
> @php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies
Lock file operations: 110 installs, 0 updates, 0 removals
- Locking brick/math (0.11.0)
- Locking dflydev/dot-access-data (v3.0.2)
- Locking doctrine/inflector (2.0.0)
- Locking doctrine/lexer (3.0.0)
- Locking dragonmantank/cron-expression (v3.3.3)
- Locking egulias/email-validator (4.0.1)
- Locking fakerphp/faker (v1.23.0)
- Locking filp/whoops (2.15.3)
- Locking fruitcake/php-cors (v1.2.0)
```

Рис.5. процесс

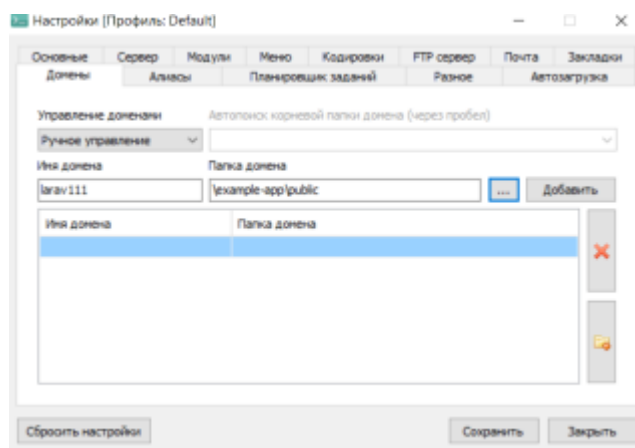


Рис.6. создание домена

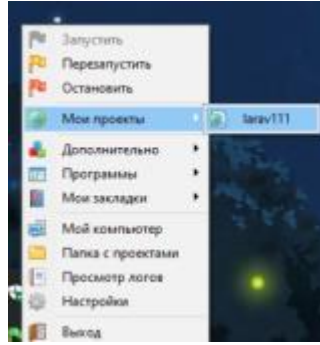


Рис.7. проверка созданного домена

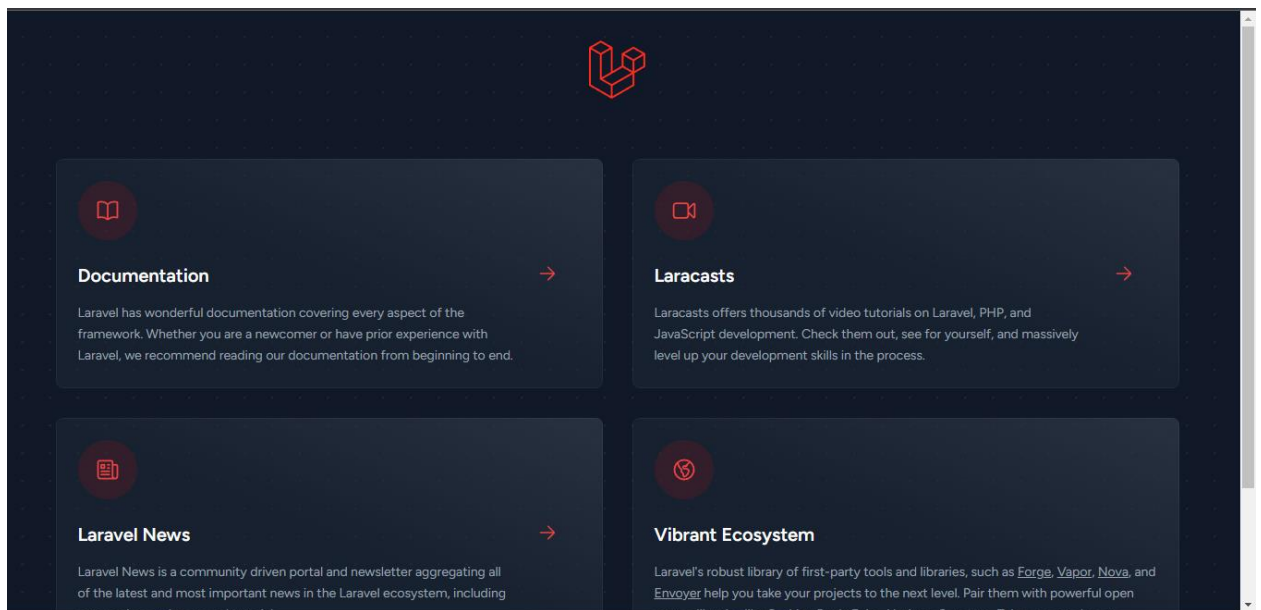


Рис.8. результат

2. CONFIGURATION

Все файлы конфигурации платформы Laravel хранятся в этом config каталоге.

Эти файлы конфигурации позволяют вам настраивать такие вещи, как информация о подключении к базе данных, информация о вашем почтовом сервере, а также различные другие основные параметры конфигурации, такие как часовой пояс вашего приложения и ключ шифрования.

Вы можете получить краткий обзор конфигурации, драйверов и среды вашего приложения с помощью about команды Artisan.

```
php artisan about
```

Рис.9. пример

Если вас интересует только определенный раздел вывода обзора приложения, вы можете отфильтровать этот раздел, используя параметр `--only`

```
php artisan about --only=environment
```

Рис.10. пример

Или, чтобы подробно изучить значения конкретного файла конфигурации, вы можете использовать `config:show` команду Artisan:

```
php artisan config:show database
```

Рис.11. пример

КОНФИГУРАЦИЯ СРЕДЫ

Чтобы использовать локально другой драйвер кэша, чем на рабочем сервере.

Чтобы упростить эту задачу, Laravel использует PHP-библиотеку [DotEnv](#). В новой установке Laravel корневой каталог вашего приложения будет содержать файл `.env.example`, который определяет множество общих переменных среды. В процессе установки Laravel этот файл будет автоматически скопирован в `.env`.

Файл Laravel по умолчанию `.env` содержит некоторые общие значения конфигурации, которые могут различаться в зависимости от того, работает ли ваше приложение локально или на рабочем веб-сервере. Эти значения затем извлекаются из различных файлов конфигурации Laravel в `config` каталоге с помощью функции `Laravel env`.

Если вы разрабатываете вместе с командой, возможно, вы захотите и дальше включать файл `.env.example` в свое приложение. Поместив значения-заполнители в пример файла конфигурации, другие разработчики из вашей команды смогут четко видеть, какие переменные среды необходимы для запуска вашего приложения.

Ваш `.env` файл не должен быть передан в систему контроля версий вашего приложения, поскольку каждому разработчику/серверу, использующему ваше приложение, может потребоваться другая конфигурация среды. Кроме того, это может представлять угрозу безопасности в случае, если

злоумышленник получит доступ к вашему хранилищу системы контроля версий, поскольку любые конфиденциальные учетные данные будут раскрыты.

Однако можно зашифровать файл среды, используя встроенное шифрование среды Laravel . Зашифрованные файлы среды можно безопасно поместить в систему контроля версий.

Прежде чем загружать переменные среды вашего приложения, Laravel определяет, APP_ENV предоставлена ли переменная среды извне или –env указан аргумент CLI. Если это так, Laravel попытается загрузить .env.[APP_ENV]файл, если он существует. Если он не существует, .env будет загружен файл по умолчанию.

ТИПЫ ПЕРЕМЕННЫХ

<code>.env</code> Value	<code>env()</code> Value
true	(bool) true
(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) "
(empty)	(string) "
null	(null) null
(null)	(null) null

Рис.12. пример

Если вам нужно определить переменную среды со значением, содержащим пробелы, вы можете сделать это, заключив значение в двойные кавычки:

```
APP_NAME="My Application"
```

Рис.3. пример

Все переменные, перечисленные в .env файле, будут загружены в \$_ENV суперглобальный PHP, когда ваше приложение получит запрос. Однако вы можете использовать эту env функцию для получения значений из этих

переменных в ваших файлах конфигурации. Фактически, если вы просмотрите файлы конфигурации Laravel, вы заметите, что многие параметры уже используют эту функцию:

```
'debug' => env('APP_DEBUG', false),
```

Рис.4. пример

Текущая среда приложения определяется через APP_ENV переменную из вашего env файла. Вы можете получить доступ к этому значению через environment метод на App facad:

```
use Illuminate\Support\Facades\App;  
  
$environment = App::environment();
```

рис.5. пример

Вы также можете передать аргументы методу environment, чтобы определить, соответствует ли среда заданному значению. Метод вернется true, если среда соответствует любому из заданных значений:

```
if (App::environment('local')) {  
    // The environment is local  
}  
  
if (App::environment(['local', 'staging'])) {  
    // The environment is either local OR staging...  
}
```

Рис.6. пример

Текущее обнаружение среды приложения можно переопределить, определив APP_ENV переменную среды уровня сервера.

ШИФРОВАНИЕ ФАЙЛОВ СРЕДЫ

Незашифрованные файлы среды никогда не должны храниться в системе контроля версий. Однако Laravel позволяет вам шифровать файлы среды, чтобы их можно было безопасно добавить в систему контроля версий вместе с остальной частью вашего приложения.

Чтобы зашифровать файл среды, вы можете использовать `env:encrypt` команду:

```
php artisan env:encrypt
```

Рис.7. пример

Выполнение `env:encrypt` команды зашифрует ваш `.env` файл и поместит зашифрованное содержимое в `.env.encrypted` файл. Ключ дешифрования представлен в выходных данных команды и должен храниться в безопасном менеджере паролей. Если вы хотите предоставить свой собственный ключ шифрования, вы можете использовать эту `--key` опцию при вызове команды:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

Рис.8. пример

Длина предоставленного ключа должна соответствовать длине ключа, требуемой используемым шифром шифрования. По умолчанию Laravel будет использовать AES-256-CBC шифр, для которого требуется ключ из 32 символов. Вы можете использовать любой шифр, поддерживаемый шифратором Laravel, передав эту `--cipher` опцию при вызове команды.

Если ваше приложение имеет несколько файлов среды, например `.env` и `.env.staging`, вы можете указать файл среды, который должен быть зашифрован, указав имя среды с помощью `--env` параметра:

```
php artisan env:encrypt --env=staging
```

Рис.9. пример

РАСШИФРОВКА

Чтобы расшифровать файл среды, вы можете использовать `env:decrypt` команду. Для этой команды требуется ключ дешифрования, который Laravel получит из `LARAVEL_ENV_ENCRYPTION_KEY` переменной среды:

```
php artisan env:decrypt
```

Рис.10. пример

Или ключ может быть предоставлен непосредственно команде с помощью —key опции:

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

Рис.11. пример

Когда env:decrypt команда вызывается, Laravel расшифровывает содержимое файла .env.encrypted и помещает расшифрованное содержимое в .env файл.

--cipher Команде может быть предоставлена опция для env:decrypt использования собственного шифра шифрования:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

Рис.12. пример

Если ваше приложение имеет несколько файлов среды, например .env и .env.staging, вы можете указать файл среды, который необходимо расшифровать, указав имя среды с помощью —env параметра:

```
php artisan env:decrypt --env=staging
```

Рис.13. пример

Чтобы перезаписать существующий файл среды, вы можете предоставить —force команде опцию env:decrypt:

```
php artisan env:decrypt --force
```

Рис.14. пример

ДОСТУП К ЗНАЧЕНИЯМ КОНФИГУРАЦИИ

Вы можете легко получить доступ к значениям конфигурации с помощью глобальной `config` функции из любого места вашего приложения. Доступ к значениям конфигурации можно получить с помощью синтаксиса «точка», который включает имя файла и параметр, к которому вы хотите получить доступ. Также можно указать значение по умолчанию, которое будет возвращено, если опция конфигурации не существует:

```
$value = config('app.timezone');

// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

Рис.15. пример

Чтобы установить значения конфигурации во время выполнения, передайте в функцию массив `config`:

```
config(['app.timezone' => 'America/Chicago']);
```

Рис.16. пример

КЭШИРОВАНИЕ КОНФИГУРАЦИИ

Чтобы повысить скорость вашего приложения, вам следует кэшировать все файлы конфигурации в один файл с помощью `config:cache` команды Artisan. Это объединит все параметры конфигурации вашего приложения в один файл, который можно будет быстро загрузить с помощью платформы.

Обычно эту команду следует запускать `php artisan config:cache` в процессе производственного развертывания. Эту команду не следует запускать во время локальной разработки, поскольку параметры конфигурации часто придется изменять в ходе разработки вашего приложения.

После кэширования конфигурации `.env` файл вашего приложения не будет загружаться платформой во время запросов или команд Artisan; следовательно, `env` функция будет возвращать только внешние переменные среды системного уровня.

По этой причине вам следует убедиться, что вы вызываете `env` функцию только из `config` файлов конфигурации () вашего приложения. Вы можете

увидеть множество примеров этого, изучив файлы конфигурации Laravel по умолчанию. Доступ к значениям конфигурации можно получить из любого места вашего приложения с помощью `config` функции, описанной выше .

Эту `config:clear` команду можно использовать для очистки кэшированной конфигурации:



```
php artisan config:clear
```

Рис.17. пример

3. DIRECTORY STRUCTURE

Структура приложения Laravel по умолчанию призвана стать отличной отправной точкой как для больших, так и для небольших приложений. Но вы можете организовать свое приложение так, как вам нравится. Laravel почти не накладывает ограничений на расположение того или иного класса — при условии, что Composer может автоматически загружать класс.

КОРНЕВОЙ КАТАЛОГ

- **Каталог приложений**

Большая часть вашего приложения находится в `app` каталоге. По умолчанию этот каталог находится в пространстве имен `App` автоматически загружается Composer с использованием стандарта автозагрузки PSR-4 .

Каталог `app` содержит множество дополнительных каталогов, таких как `Console`, `Http Providers`. Считайте, что каталоги `Console` и `Http` предоставляют API для ядра вашего приложения. Протокол HTTP и CLI являются механизмами взаимодействия с вашим приложением, но фактически не содержат логики приложения. Другими словами, это два способа подачи команд вашему приложению. В `Console` каталоге содержатся все ваши команды `Artisan`, а в `Http` каталоге — ваши контроллеры, промежуточное ПО и запросы.

Внутри каталога будет создано множество других каталогов, `app` когда вы используете `make` команды `Artisan` для создания классов. Так, например, `app/Jobs` каталог не будет существовать, пока вы не выполните `make:job` команду `Artisan` для создания класса задания.

- **Каталог начальной загрузки**

Каталог bootstrap содержит app.php файл, который загружает фреймворк. В этом каталоге также находится cache каталог, содержащий файлы, созданные платформой для оптимизации производительности, такие как файлы кэша маршрутов и служб. Обычно вам не нужно изменять какие-либо файлы в этом каталоге

- **Каталог конфигурации**

Каталог config, как следует из названия, содержит все файлы конфигурации вашего приложения. Будет хорошей идеей прочитать все эти файлы и ознакомиться со всеми доступными вам опциями.

- **Каталог базы данных**

Каталог database содержит миграции вашей базы данных, фабрики моделей и исходные данные. При желании вы также можете использовать этот каталог для хранения базы данных SQLite.

- **Публичный каталог**

В public каталоге находится index.php файл, который является точкой входа для всех запросов, поступающих в ваше приложение, и настраивает автозагрузку. В этом каталоге также хранятся ваши ресурсы, такие как изображения, JavaScript и CSS.

- **Каталог ресурсов**

Каталог resources содержит ваши представления, а также необработанные, некомпилированные ресурсы, такие как CSS или JavaScript.

- **Каталог маршрутов**

Каталог routes содержит все определения маршрутов для вашего приложения. По умолчанию в Laravel включено несколько файлов маршрутов: web.php, api.php, console.php и channels.php.

Файл web.php содержит маршруты, которые RouteServiceProvider помещаются в web группу промежуточного программного обеспечения, которая обеспечивает состояние сеанса, защиту CSRF и шифрование файлов cookie. Если ваше приложение не предлагает RESTful API без сохранения состояния, то все ваши маршруты, скорее всего, будут определены в файле web.php.

Файл api.php содержит маршруты, которые RouteServiceProvider помещаются в api группу промежуточного программного обеспечения. Эти маршруты

предназначены для сохранения состояния, поэтому запросы, поступающие в приложение через эти маршруты, предназначены для аутентификации с помощью токенов и не будут иметь доступа к состоянию сеанса.

В этом `console.php` файле вы можете определить все консольные команды, основанные на закрытии. Каждое замыкание привязано к экземпляру команды, что обеспечивает простой подход к взаимодействию с методами ввода-вывода каждой команды. Несмотря на то, что этот файл не определяет маршруты HTTP, он определяет консольные точки входа (маршруты) в ваше приложение.

В этом `channels.php` файле вы можете зарегистрировать все каналы трансляции событий, которые поддерживает ваше приложение.

- **Каталог хранения**

Каталог `storage` содержит ваши журналы, скомпилированные шаблоны Blade, сеансы на основе файлов, кэши файлов и другие файлы, созданные платформой. Этот каталог разделен на каталоги `app`, `framework` и `logs`. Этот `app` каталог можно использовать для хранения любых файлов, созданных вашим приложением. Каталог `framework` используется для хранения файлов и кешей, созданных платформой. Наконец, `logs` каталог содержит файлы журналов вашего приложения.

Каталог `storage/app/public` может использоваться для хранения файлов, созданных пользователем, таких как аватары профилей, которые должны быть общедоступными. Вам следует создать символическую ссылку, `public/storage` указывающую на этот каталог. Вы можете создать ссылку с помощью `php artisan storage:link` команды Artisan.

- **Каталог тестов**

Каталог `tests` содержит ваши автоматические тесты. Примеры модульных тестов и тестов функций PHPUnit предоставляются «из коробки». Каждый тестовый класс должен иметь суффикс `Test`. Вы можете запускать тесты с помощью команд `phpunit` или `php vendor/bin/phpunit`. Или, если вам нужно более подробное и красивое представление результатов вашего теста, вы можете запустить тесты с помощью `php artisan test` команды Artisan.

- **Каталог поставщиков**

Каталог `vendor` содержит ваши зависимости Composer .

- **Каталог вещаний**

Каталог `Broadcasting` содержит все классы широковещательных каналов для вашего приложения. Эти классы генерируются с помощью `make:channel`

команды. Этот каталог не существует по умолчанию, но он будет создан для вас при создании первого канала. Чтобы узнать больше о каналах, ознакомьтесь с документацией по трансляции событий.

- **Каталог консоли**

Каталог Console содержит все пользовательские команды Artisan для вашего приложения. Эти команды могут быть сгенерированы с помощью `make:command` команды. В этом каталоге также находится ядро вашей консоли, в котором регистрируются ваши пользовательские команды Artisan и определяются запланированные задачи.

- **Каталог событий**

По умолчанию этот каталог не существует, но он будет создан для вас командами `Artisan event:generate` и `make:event`. В Events каталоге хранятся классы событий. События могут использоваться для оповещения других частей вашего приложения о том, что произошло определенное действие, обеспечивая большую гибкость и развязку.

- **Каталог исключений**

Каталог Exceptions содержит обработчик исключений вашего приложения, а также является хорошим местом для размещения любых исключений, создаваемых вашим приложением. Если вы хотите настроить способ регистрации и обработки исключений, вам следует изменить `Handler` класс в этом каталоге.

- **HTTP каталог**

Каталог Http содержит ваши контроллеры, промежуточное программное обеспечение и запросы форм. Почти вся логика обработки запросов, поступающих в ваше приложение, будет размещена в этом каталоге.

- **Каталог вакансий**

По умолчанию этот каталог не существует, но он будет создан для вас, если вы выполните `make:job` команду Artisan. В Jobs каталоге хранятся задания вашего приложения, помещаемые в очередь. Задания могут быть поставлены в очередь вашим приложением или выполняться синхронно в рамках текущего жизненного цикла запроса. Задания, которые выполняются синхронно во время текущего запроса, иногда называют «командами», поскольку они являются реализацией шаблона команды.

- **Каталог слушателей**

Этот каталог не существует по умолчанию, но он будет создан для вас, если вы выполните команды `event:generate`

или `make:listenerArtisan`. Каталог `Listeners` содержит классы, которые обрабатывают ваши события. Прослушиватели событий получают экземпляр события и выполняют логику в ответ на инициируемое событие. Например, `UserRegistered` событие может обрабатываться прослушивателем `SendWelcomeEmail`.

- **Почтовый каталог**

По умолчанию этот каталог не существует, но он будет создан для вас, если вы выполните `make:mail` команду Artisan. Каталог `Mail` содержит все ваши классы, которые представляют электронные письма, отправленные вашим приложением. Объекты `Mail` позволяют инкапсулировать всю логику создания электронного письма в одном простом классе, который можно отправить с помощью этого `Mail::send` метода.

- **Каталог моделей**

Каталог `Models` содержит все ваши классы моделей Eloquent . Eloquent ORM, входящий в состав Laravel, предоставляет красивую и простую реализацию `ActiveRecord` для работы с вашей базой данных. Каждая таблица базы данных имеет соответствующую «модель», которая используется для взаимодействия с этой таблицей. Модели позволяют запрашивать данные в таблицах, а также вставлять в таблицу новые записи.

- **Каталог уведомлений**

По умолчанию этот каталог не существует, но он будет создан для вас, если вы выполните `make:notification` команду Artisan. Каталог `Notifications` содержит все «транзакционные» уведомления, отправляемые вашим приложением, например, простые уведомления о событиях, происходящих внутри вашего приложения. Функция уведомлений Laravel абстрагирует отправку уведомлений с помощью различных драйверов, таких как электронная почта, Slack, SMS или сохранение в базе данных.

- **Каталог политик**

По умолчанию этот каталог не существует, но он будет создан для вас, если вы выполните `make:policy` команду Artisan. Каталог `Policies` содержит классы политики авторизации для вашего приложения. Политики используются для определения того, может ли пользователь выполнить данное действие с ресурсом.

- **Каталог правил**

По умолчанию этот каталог не существует, но он будет создан для вас, если вы выполните `make:rule` команду Artisan. Каталог `Rules` содержит объекты пользовательских правил проверки для вашего приложения. Правила

используются для инкапсуляции сложной логики проверки в простой объект. Для получения дополнительной информации ознакомьтесь с документацией по валидации.

4. FRONTEND

Введение

Laravel — это серверная среда, которая предоставляет все функции, необходимые для создания современных веб-приложений, такие как маршрутизация, проверка, кэширование, очереди, хранилище файлов и многое другое. Тем не менее, мы считаем важным предложить разработчикам прекрасный комплексный опыт, включая мощные подходы для создания внешнего интерфейса вашего приложения.

Существует два основных способа разработки внешнего интерфейса при создании приложения с помощью Laravel, и какой подход вы выберете, зависит от того, хотите ли вы создавать свой внешний интерфейс с использованием PHP или с использованием фреймворков JavaScript, таких как Vue и React. Ниже мы обсудим оба этих варианта, чтобы вы могли принять обоснованное решение относительно наилучшего подхода к разработке внешнего интерфейса для вашего приложения.

Использование PHP

PHP и Blade

В прошлом большинство приложений PHP отображали HTML в браузере, используя простые шаблоны HTML, перемежающиеся echo-операторами PHP, которые отображают данные, полученные из базы данных во время запроса:

```
<div>
    <?php foreach ($users as $user): ?>
        Hello, <?php echo $user->name; ?> <br />
    <?php endforeach; ?>
</div>
```

Рис.18 пример

В Laravel такого подхода к рендерингу HTML все еще можно достичь с помощью представлений и Blade. Blade — чрезвычайно легкий язык шаблонов, который предоставляет удобный короткий синтаксис для отображения данных, перебора данных и многого другого:

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

Рис.19 пример

При создании приложений таким образом отправка форм и другие взаимодействия со страницами обычно получают совершенно новый HTML-документ с сервера, и вся страница повторно отображается браузером. Даже сегодня многие приложения могут идеально подходить для создания интерфейсов таким образом с использованием простых шаблонов Blade.

Ожидания

Однако по мере того, как ожидания пользователей в отношении веб-приложений стали более зрелыми, многие разработчики обнаружили необходимость создания более динамичных интерфейсов с более совершенным взаимодействием. В свете этого некоторые разработчики решают начать создание интерфейса своего приложения с использованием фреймворков JavaScript, таких как Vue и React.

Другие, предпочитая использовать тот язык серверной части, который им удобен, разработали решения, которые позволяют создавать современные пользовательские интерфейсы веб-приложений, в то же время в основном используя выбранный ими серверный язык. Например, в экосистеме Rails это подстегнуло создание таких библиотек, как Turbo Hotwire и Stimulus.

В экосистеме Laravel необходимость создания современных динамических интерфейсов с использованием преимущественно PHP привела к созданию Laravel Livewire и Alpine.js.

Livewire

Laravel Livewire — это платформа для создания интерфейсов на базе Laravel, которые кажутся динамичными, современными и живыми, точно так же, как интерфейсы, созданные с использованием современных фреймворков JavaScript, таких как Vue и React.

При использовании Livewire вы создадите «компоненты» Livewire, которые отображают дискретную часть вашего пользовательского интерфейса и предоставляют методы и данные, которые можно вызывать и с которыми можно взаимодействовать из внешнего интерфейса вашего приложения. Например, простой компонент «Счетчик» может выглядеть следующим образом:

```

<?php

namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
    {
        return view('livewire.counter');
    }
}

```

Рис.20 пример

И соответствующий шаблон для счетчика будет записан так:

```

<div>
    <button wire:click="increment">+</button>
    <h1>{{ $count }}</h1>
</div>

```

Рис.21 пример

Как видите, Livewire позволяет вам писать новые атрибуты HTML, например, `wire:click` которые соединяют интерфейс и серверную часть вашего приложения Laravel. Кроме того, вы можете визуализировать текущее состояние вашего компонента, используя простые выражения Blade.

Для многих Livewire произвел революцию в разработке внешнего интерфейса с помощью Laravel, позволив им оставаться в комфортной среде Laravel при создании современных, динамичных веб-приложений. Обычно разработчики, использующие Livewire, также используют Alpine.js для

«добавления» JavaScript в свой интерфейс только там, где это необходимо, например, для визуализации диалогового окна.

Если вы новичок в Laravel, мы рекомендуем ознакомиться с основами использования представлений и Blade. Затем обратитесь к официальной документации Laravel Livewire, чтобы узнать, как вывести ваше приложение на новый уровень с помощью интерактивных компонентов Livewire.

Использование Vue/React

Однако без дополнительных инструментов объединение Laravel с Vue или React оставило бы нам необходимость решать множество сложных проблем, таких как маршрутизация на стороне клиента, гидратация данных и аутентификация. Маршрутизация на стороне клиента часто упрощается за счет использования самоуверенных фреймворков Vue/React, таких как Nuxt и Next; однако гидратация данных и аутентификация остаются сложными и громоздкими проблемами, которые необходимо решить при объединении серверной инфраструктуры, такой как Laravel, с этими интерфейсными платформами.

Inertia

К счастью, Laravel предлагает лучшее из обоих миров. Inertia устраняет разрыв между вашим приложением Laravel и вашим современным интерфейсом Vue или React, позволяя вам создавать полноценные современные интерфейсы с использованием Vue или React, одновременно используя маршруты и контроллеры Laravel для маршрутизации, гидратации данных и аутентификации — и все это в одном коде. хранилище. При таком подходе вы сможете пользоваться всей мощностью Laravel и Vue/React, не нанося вреда возможностям ни одного из инструментов.

После установки Inertia в ваше приложение Laravel вы будете писать маршруты и контроллеры, как обычно. Однако вместо возврата шаблона Blade с вашего контроллера вы вернете страницу Inertia:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Inertia\Inertia;
use Inertia\Response;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): Response
    {
        return Inertia::render('Users/Profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Рис.22 пример

Страница Inertia соответствует компоненту Vue или React, который обычно хранится в resources/js/Pages каталоге вашего приложения. Данные, переданные странице через Inertia::render метод, будут использоваться для гидратации «реквизита» компонента страницы:

```

<script setup>
import Layout from '@/Layouts/Authenticated.vue';
import { Head } from '@inertiajs/inertia-vue3';

const props = defineProps(['user']);
</script>

<template>
  <Head title="User Profile" />

  <Layout>
    <template #header>
      <h2 class="font-semibold text-xl text-gray-800 leading-tight">
        Profile
      </h2>
    </template>

    <div class="py-12">
      Hello, {{ user.name }}
    </div>
  </Layout>
</template>

```

Рис.23 пример

Серверный рендеринг

Если вы беспокоитесь о погружении в Inertia, поскольку ваше приложение требует рендеринга на стороне сервера, не волнуйтесь. Inertia предлагает поддержку рендеринга на стороне сервера. А при развертывании вашего приложения через Laravel Forge очень легко убедиться, что процесс рендеринга на стороне сервера Inertia всегда работает.

Объединение активов

По умолчанию Laravel использует Vite для объединения ваших ресурсов. Vite обеспечивает молниеносное время сборки и практически мгновенную горячую замену модулей (HMR) во время локальной разработки. Во всех новых приложениях Laravel, включая те, которые используют наши стартовые наборы, вы найдете vite.config.js файл, который загружает наш легкий плагин Laravel Vite, который делает использование Vite с приложениями Laravel приятным.

Самый быстрый способ начать работу с Laravel и Vite — начать разработку вашего приложения с помощью Laravel Breeze, нашего простейшего стартового набора, который запускает ваше приложение, предоставляя механизмы аутентификации внешнего и внутреннего интерфейса.

5. СТАРТОВЫЕ НАБОРЫ

Laravel Breeze

Laravel Breeze — это минимальная и простая реализация всех функций аутентификации Laravel, включая вход в систему, регистрацию, сброс пароля, проверку электронной почты и подтверждение пароля. Кроме того, Breeze включает простую страницу «профиля», на которой пользователь может обновить свое имя, адрес электронной почты и пароль.

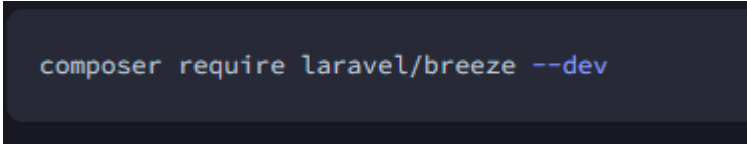
Слой представления Laravel Breeze по умолчанию состоит из простых шаблонов Blade, оформленных с помощью Tailwind CSS. Или Breeze может создать ваше приложение с помощью Vue или React и Inertia.

Breeze обеспечивает прекрасную отправную точку для создания нового приложения Laravel, а также является отличным выбором для проектов, которые планируют вывести свои шаблоны Blade на новый уровень с помощью Laravel Livewire.

Если вы новичок в Laravel, смело присоединяйтесь к Laravel Bootcamp. Laravel Bootcamp проведет вас через создание вашего первого приложения Laravel с помощью Breeze. Это отличный способ познакомиться со всем, что могут предложить Laravel и Breeze.

Установка

Сначала вам следует создать новое приложение Laravel, настроить базу данных и выполнить миграцию базы данных. После создания нового приложения Laravel вы можете установить Laravel Breeze с помощью Composer:



```
composer require laravel/breeze --dev
```

Рис.24 пример

После того, как Composer установил пакет Laravel Breeze, вы можете запустить breeze:install команду Artisan. Эта команда публикует представления аутентификации, маршруты, контроллеры и другие ресурсы в вашем приложении. Laravel Breeze публикует весь свой код в вашем приложении, чтобы вы имели полный контроль над его функциями и реализацией.

Команда `breeze:install` предложит вам выбрать предпочитаемый стек интерфейса и среду тестирования:

```
php artisan breeze:install

php artisan migrate
npm install
npm run dev
```

Рис.25 пример

Breeze и Next.js/API

Laravel Breeze также может создать API аутентификации, готовый аутентифицировать современные приложения JavaScript, например, на базе Next, Nuxt и других. Чтобы начать, выберите стек API в качестве желаемого при выполнении `breeze:install` команды Artisan:

```
php artisan breeze:install

php artisan migrate
```

Рис.26 пример

Во время установки Breeze добавит `FRONTEND_URL` переменную среды в файл вашего приложения. `env`. Этот URL-адрес должен быть URL-адресом вашего приложения JavaScript. Обычно это происходит `http://localhost:3000` во время локальной разработки. Кроме того, вам следует убедиться, что для вас `APP_URL` установлено значение `http://localhost:8000`, которое является URL-адресом по умолчанию, используемым `serve` командой Artisan.

Наконец, вы готовы соединить этот бэкэнд с внешним интерфейсом по вашему выбору. Эталонная реализация интерфейса Breeze Next доступна на GitHub. Этот интерфейс поддерживается Laravel и содержит тот же пользовательский интерфейс, что и традиционные стеки Blade и Inertia, предоставляемые Breeze.

Laravel Jetstream

В то время как Laravel Breeze обеспечивает простую и минимальную отправную точку для создания приложения Laravel, Jetstream дополняет эту функциональность более надежными функциями и дополнительными стеками интерфейсных технологий. Новичкам в Laravel мы рекомендуем изучить основы Laravel Breeze, прежде чем переходить на Laravel Jetstream.

Jetstream предоставляет красиво оформленную платформу приложений для Laravel и включает в себя вход в систему, регистрацию, проверку электронной почты, двухфакторную аутентификацию, управление сессиями, поддержку API через Laravel Sanctum и дополнительное управление командой. Jetstream разработан с использованием Tailwind CSS и предлагает на ваш выбор интерфейсные леса Livewire или Inertia.

Полную документацию по установке Laravel Jetstream можно найти в официальной документации Jetstream.

6. РАЗВЕРТЫВАНИЕ

Фреймворк Laravel имеет несколько системных требований. Вы должны убедиться, что ваш веб-сервер имеет следующую минимальную версию PHP и расширения:

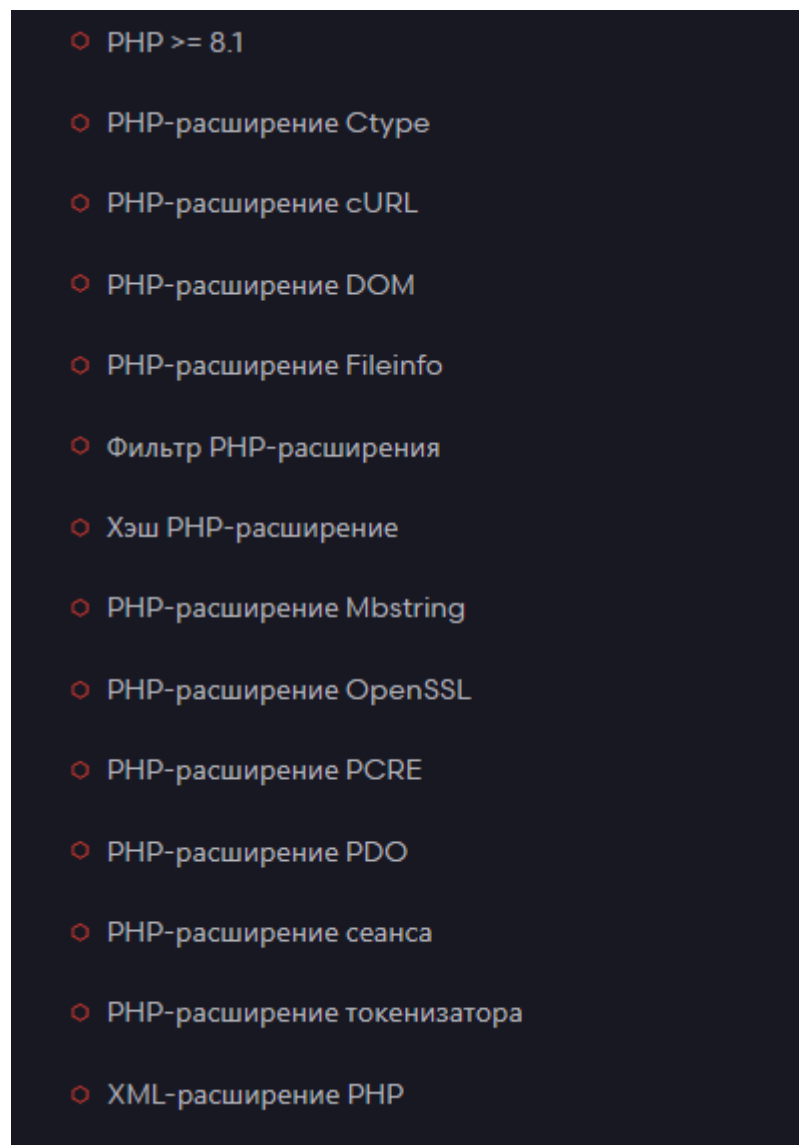


Рис.27 пример

Конфигурация сервера

Ngnx

Если вы развертываете свое приложение на сервере под управлением Nginx, вы можете использовать следующий файл конфигурации в качестве отправной точки для настройки вашего веб-сервера. Скорее всего, этот файл потребуется настроить в зависимости от конфигурации вашего сервера. Если вам нужна помощь в управлении вашим сервером, рассмотрите возможность использования сторонней службы управления и развертывания серверов Laravel, такой как Laravel Forge.

Убедитесь, что ваш веб-сервер, как и в конфигурации ниже, направляет все запросы к `public/index.php` файлу вашего приложения. Никогда не пытайтесь переместить `index.php` файл в корень вашего проекта, поскольку обслуживание приложения из корня проекта приведет к тому, что многие конфиденциальные файлы конфигурации будут доступны в общедоступном Интернете:

```
server {
    listen 80;
    listen [::]:80;
    server_name example.com;
    root /srv/example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-Content-Type-Options "nosniff";

    index index.php;

    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ /\.php$ {
        fastcgi_pass unix:/var/run/php/php8.1-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
    }

    location ~ /\.(!well-known).* {
        deny all;
    }
}
```

Рис.28 пример

Оптимизация

Оптимизация автозагрузчика

При развертывании в рабочей среде убедитесь, что вы оптимизируете карту автозагрузчика классов Composer, чтобы Composer мог быстро найти подходящий файл для загрузки для данного класса:

```
composer install --optimize-autoloader --no-dev
```

Рис.29 пример

Помимо оптимизации автозагрузчика, вы всегда должны обязательно включать файл `composer.lock` в репозиторий системы контроля версий вашего проекта. Зависимости вашего проекта можно установить гораздо быстрее, если файл `composer.lock` присутствует.

Конфигурация кэширования

При развертывании вашего приложения в рабочей среде вы должны убедиться, что вы запускаете `config:cache` команду Artisan во время процесса развертывания:

```
php artisan config:cache
```

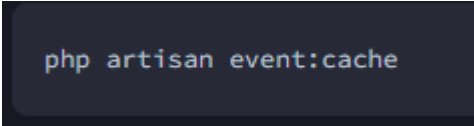
Рис.30 пример

Эта команда объединит все файлы конфигурации Laravel в один кэшированный файл, что значительно сокращает количество обращений, которые платформа должна совершить к файловой системе при загрузке значений вашей конфигурации.

Если вы выполняете `config:cache` команду во время процесса развертывания, вы должны быть уверены, что вызываете функцию только `env` из файлов конфигурации. После кэширования конфигурации `.env` файл не будет загружен, и все вызовы `env` функции для `.env` переменных будут возвращать значение `null`.

Кэширование событий

Если ваше приложение использует обнаружение событий, вам следует кэшировать события вашего приложения для сопоставлений прослушивателей во время процесса развертывания. Это можно сделать, вызвав `event:cache` команду Artisan во время развертывания:



```
php artisan event:cache
```

Рис.31 пример

Кэширование представлений

При развертывании вашего приложения в рабочей среде вы должны убедиться, что вы запускаете `view:cache` команду Artisan во время процесса развертывания:

Режим отладки

Параметр отладки в вашем файле конфигурации `config/app.php` определяет, какой объем информации об ошибке фактически отображается пользователю. По умолчанию этот параметр настроен на уважение значения переменной `APP_DEBUG` среды, которая хранится в `.env` файле вашего приложения.

В вашей производственной среде это значение всегда должно быть `false`. Если для `APP_DEBUG` переменной установлено значение «`true` в производстве», вы рискуете раскрыть конфиденциальные значения конфигурации конечным пользователям вашего приложения.

Простое развертывание с помощью Forge / Vapor

Laravel Forge

Если вы не совсем готовы управлять конфигурацией собственного сервера или вам неудобно настраивать все различные службы, необходимые для запуска надежного приложения Laravel, Laravel Forge — прекрасная альтернатива.

Laravel Forge может создавать серверы на базе различных поставщиков инфраструктуры, таких как DigitalOcean, Linode, AWS и других. Кроме того, Forge устанавливает и управляет всеми инструментами, необходимыми для создания надежных приложений Laravel, такими как Nginx, MySQL, Redis, Memcached, Beanstalk и другие.

Laravel Vapor

Если вам нужна полностью бессерверная платформа развертывания с автоматическим масштабированием, настроенная для Laravel, обратите внимание на Laravel Vapor. Laravel Vapor — это бессерверная платформа развертывания Laravel на базе AWS. Запустите свою инфраструктуру Laravel на Vapor и влюбитесь в масштабируемую простоту бессерверных технологий. Laravel Vapor настроен создателями Laravel для беспрепятственной работы с инфраструктурой, поэтому вы можете продолжать писать свои приложения Laravel точно так же, как вы привыкли.

7. Жизненный цикл запроса

Первые шаги

Точкой входа для всех запросов к приложению Laravel является файл `public/index.php`. Все запросы направляются в этот файл конфигурацией вашего веб-сервера (Apache/Nginx). Файл `index.php` не содержит много кода. Скорее, это отправная точка для загрузки остальной части платформы.

Файл `index.php` загружает определение автозагрузчика, сгенерированное Composer, а затем извлекает экземпляр приложения Laravel из `bootstrap/app.php`. Первое действие, предпринимаемое самим Laravel, — это создание экземпляра контейнера приложения/сервиса.

HTTP/консольные ядра

Далее входящий запрос отправляется либо в ядро HTTP, либо в ядро консоли, в зависимости от типа запроса, поступающего в приложение. Эти два ядра служат центральным расположением, через которое проходят все запросы. А пока давайте сосредоточимся на ядре HTTP, которое находится в `app/Http/Kernel.php`.

Ядро HTTP расширяет `Illuminate\Foundation\Http\Kernel` класс, который определяет массив, `bootstrappers` который будет запущен перед выполнением запроса. Эти загрузчики настраивают обработку ошибок, настраивают ведение журнала, определяют среду приложения и выполняют другие задачи, которые необходимо выполнить до фактической обработки запроса. Обычно эти классы обрабатывают внутреннюю конфигурацию Laravel, о которой вам не нужно беспокоиться

Ядро HTTP также определяет список промежуточного программного обеспечения HTTP, через которое должны пройти все запросы, прежде чем они будут обработаны приложением. Это промежуточное программное обеспечение обрабатывает чтение и запись сеанса HTTP, определяет, находится ли приложение в режиме обслуживания, проверяет токен CSRF и многое другое. Мы поговорим об этом подробнее в ближайшее время.

Сигнатура метода ядра HTTP `handle` довольно проста: он получает `Request` и возвращает `Response`. Думайте о ядре как о большом черном ящике, в котором представлено все ваше приложение. Отправьте ему HTTP-запросы, и он вернет HTTP-ответы.

Поставщики услуг

Одним из наиболее важных действий при загрузке ядра является загрузка поставщиков услуг для вашего приложения. Поставщики услуг несут ответственность за загрузку всех различных компонентов платформы, таких как база данных, очередь, компоненты проверки и маршрутизации. Все поставщики услуг для приложения настраиваются в массиве `config/app.php` файла конфигурации `providers`.

Laravel будет перебирать этот список провайдеров и создавать экземпляры каждого из них. После создания экземпляров поставщиков `register` метод будет вызываться для всех поставщиков. Затем, как только все поставщики будут зарегистрированы, `boot` метод будет вызываться для каждого поставщика. Это связано с тем, что поставщики услуг могут зависеть от того, что каждая привязка контейнера зарегистрирована и доступна к моменту `boot` выполнения их метода.

По сути, каждая основная функция, предлагаемая Laravel, загружается и настраивается поставщиком услуг. Поскольку они загружают и настраивают множество функций, предлагаемых инфраструктурой, поставщики услуг являются наиболее важным аспектом всего процесса загрузки Laravel.

Маршрутизация

Одним из наиболее важных поставщиков услуг в вашем приложении является платформа `App\Providers\RouteServiceProvider`. Этот поставщик услуг загружает файлы маршрутов, содержащиеся в `routes` каталоге вашего приложения. Давайте, взломайте `RouteServiceProvider` код и посмотрите, как он работает!

После загрузки приложения и регистрации всех поставщиков услуг оно `Request` будет передано маршрутизатору для отправки. Маршрутизатор отправит запрос на маршрут или контроллер, а также запустит любое промежуточное программное обеспечение, специфичное для маршрута.

Промежуточное ПО предоставляет удобный механизм фильтрации или проверки HTTP-запросов, входящих в ваше приложение. Например, Laravel включает промежуточное программное обеспечение, которое проверяет, аутентифицирован ли пользователь вашего приложения. Если пользователь не аутентифицирован, промежуточное программное обеспечение перенаправит пользователя на экран входа в систему. Однако если пользователь аутентифицирован, промежуточное программное обеспечение позволит запросу продолжить работу в приложении. Некоторое промежуточное программное обеспечение назначается всем маршрутам в приложении, например, определенным в свойстве `$middleware` вашего ядра HTTP, тогда как некоторые назначаются только определенным маршрутам

или группам маршрутов. Вы можете узнать больше о промежуточном программном обеспечении, прочитав полную документацию по промежуточному программному обеспечению.

Если запрос проходит через все назначенное промежуточное программное обеспечение соответствующего маршрута, метод маршрута или контроллера будет выполнен, а ответ, возвращенный методом маршрута или контроллера, будет отправлен обратно через цепочку промежуточного программного обеспечения маршрута.

Заканчивать

Как только метод маршрута или контроллера вернет ответ, ответ отправится обратно через промежуточное программное обеспечение маршрута, давая приложению возможность изменить или проверить исходящий ответ.

Наконец, как только ответ проходит обратно через промежуточное программное обеспечение, метод ядра HTTP `handle` возвращает объект ответа, и `index.php` файл вызывает `send` метод возвращенного ответа. Метод `send` отправляет содержимое ответа в веб-браузер пользователя. Мы завершили путешествие по всему жизненному циклу запроса Laravel!

Сосредоточьтесь на поставщиках услуг

Поставщики услуг действительно являются ключом к загрузке приложения Laravel. Экземпляр приложения создается, поставщики услуг регистрируются, и запрос передается загруженному приложению. Это действительно так просто!

Очень важно иметь четкое представление о том, как приложение Laravel создается и загружается через поставщиков услуг. В этом каталоге хранятся поставщики услуг вашего приложения по умолчанию `app/Providers`.

По умолчанию он `AppServiceProvider` довольно пуст. Этот поставщик — отличное место для добавления собственной начальной загрузки вашего приложения и привязок сервисного контейнера. Для больших приложений вам может потребоваться создать несколько поставщиков услуг, каждый из которых будет иметь более детальную загрузку для конкретных служб, используемых вашим приложением.

8. СЕРВИСНЫЙ КОНТЕЙНЕР

Сервисный контейнер Laravel — это мощный инструмент для управления зависимостями классов и внедрения зависимостей. Внедрение зависимостей — это причудливая фраза, которая по сути означает следующее: зависимости класса «вводятся» в класс через конструктора или, в некоторых случаях, методы «установки».

Давайте посмотрим на простой пример:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;
use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Show the profile for the given user.
     */
    public function show(string $id): View
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

Рис. 32 пример

В этом примере UserController необходимо получить пользователей из источника данных. Итак, мы добавим сервис, который сможет извлекать пользователей. В этом контексте мы, UserRepository скорее всего, используем Eloquent для получения информации о пользователе из базы данных. Однако, поскольку репозиторий внедрен, мы можем легко заменить его другой реализацией. Мы также можем легко «издеваться» или создать фиктивную реализацию UserRepository при тестировании нашего приложения.

Глубокое понимание сервисного контейнера Laravel необходимо для создания мощного и большого приложения, а также для внесения вклада в само ядро Laravel.

Разрешение нулевой конфигурации

Если класс не имеет зависимостей или зависит только от других конкретных классов (а не от интерфейсов), контейнеру не нужно указывать, как разрешить этот класс. Например, вы можете поместить в свой routes/web.php файл следующий код:

```
<?php

class Service
{
    // ...
}

Route::get('/', function (Service $service) {
    die(get_class($service));
});
```

Рис. 33 пример

В этом примере нажатие на маршрут вашего приложения /автоматически разрешит Service класс и внедрит его в обработчик вашего маршрута. Это меняет правила игры. Это означает, что вы можете разрабатывать свое приложение и использовать преимущества внедрения зависимостей, не беспокоясь о раздутых файлах конфигурации.

К счастью, многие классы, которые вы будете писать при создании приложения Laravel, автоматически получают свои зависимости через контейнер, включая контроллеры, прослушватели событий, промежуточное ПО и многое другое. Кроме того, вы можете указать зависимости в handle методе заданий в очереди. Как только вы почувствуете мощь

автоматического внедрения зависимостей с нулевой конфигурацией, вы почувствуете, что разработка без них невозможна

Когда использовать контейнер

Благодаря нулевому разрешению конфигурации вы часто будете указывать зависимости от маршрутов, контроллеров, прослушивателей событий и т. д., даже не взаимодействуя с контейнером вручную. Например, вы можете ввести подсказку для `Illuminate\Http\Request` объекта в определении маршрута, чтобы можно было легко получить доступ к текущему запросу. Несмотря на то, что нам никогда не приходится взаимодействовать с контейнером для написания этого кода, он управляет внедрением этих зависимостей «за кулисами»:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

Рис.34 пример

Во многих случаях, благодаря автоматическому внедрению зависимостей и фасадам, вы можете создавать приложения Laravel без необходимости вручную привязывать или разрешать что-либо из контейнера. Итак, когда же вы когда-нибудь будете вручную взаимодействовать с контейнером? Давайте рассмотрим две ситуации.

Во-первых, если вы пишете класс, реализующий интерфейс, и хотите указать тип этого интерфейса в конструкторе маршрута или класса, вы должны сообщить контейнеру, как разрешить этот интерфейс. Во-вторых, если вы пишете пакет Laravel, которым планируете поделиться с другими разработчиками Laravel, вам может потребоваться привязать сервисы вашего пакета к контейнеру.

Связывание

Основные привязки

Простые привязки

Почти все привязки вашего сервисного контейнера будут зарегистрированы в сервис-провайдерах, поэтому в большинстве этих примеров будет показано использование контейнера в этом контексте.

В рамках поставщика услуг у вас всегда есть доступ к контейнеру через `$this->app` свойство. Мы можем зарегистрировать привязку с помощью `bind`

метода, передав имя класса или интерфейса, который мы хотим зарегистрировать, вместе с замыканием, которое возвращает экземпляр класса:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Рис.35. пример

Обратите внимание, что мы получаем сам контейнер в качестве аргумента преобразователю. Затем мы можем использовать контейнер для разрешения дополнительных зависимостей объекта, который мы создаем.

Как уже упоминалось, обычно вы будете взаимодействовать с контейнером внутри поставщиков услуг; однако, если вы хотите взаимодействовать с контейнером вне поставщика услуг, вы можете сделать это через App facade:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

Рис.36. пример

Вы можете использовать этот `bindIf` метод для регистрации привязки контейнера, только если привязка для данного типа еще не зарегистрирована:

```
$this->app->bindIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Рис.37. пример

Привязка синглтона

Этот `singleton` метод привязывает класс или интерфейс к контейнеру, который должен быть разрешен только один раз. После разрешения одноэлементной привязки тот же экземпляр объекта будет возвращен при последующих вызовах контейнера:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->singleton(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Рис.38. пример

Вы можете использовать этот `singletonIf` метод для регистрации привязки одноэлементного контейнера, только если привязка для данного типа еще не зарегистрирована

```
$this->app->singletonIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Рис.39. пример

Привязка синглов с ограниченной областью действия

Этот `scoped` метод привязывает класс или интерфейс к контейнеру, который должен быть разрешен только один раз в течение данного жизненного цикла запроса/задания Laravel. Хотя этот метод похож на этот `singleton` метод, экземпляры, зарегистрированные с использованием этого `scoped` метода, будут сбрасываться всякий раз, когда приложение Laravel начинает новый «жизненный цикл», например, когда исполнитель Laravel Octane обрабатывает новый запрос или когда работник очереди Laravel обрабатывает новое задание:

```

use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});

```

Рис.40. пример

Экземпляр привязки

Вы также можете привязать существующий экземпляр объекта к контейнеру, используя этот `instance` метод. Данный экземпляр всегда будет возвращаться при последующих вызовах контейнера:

```

use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);

```

Рис.41. пример

Привязка интерфейсов к реализациям

Очень мощной функцией сервисного контейнера является его способность привязывать интерфейс к заданной реализации. Например, предположим, что у нас есть `EventPusher` интерфейс и `RedisEventPusher` реализация. После того, как мы запрограммировали `RedisEventPusher` реализацию этого интерфейса, мы можем зарегистрировать его в сервисном контейнере следующим образом:

```

use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);

```

Рис.42. пример

Этот оператор сообщает контейнеру, что он должен внедрить, RedisEventPusher когда классу требуется реализация EventPusher. Теперь мы можем указать EventPusher интерфейс в конструкторе класса, который разрешается контейнером. Помните, что контроллеры, прослушиватели событий, промежуточное ПО и различные другие типы классов в приложениях Laravel всегда разрешаются с использованием контейнера:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 */
public function __construct(
    protected EventPusher $pusher
) {}
```

Рис.43. пример

Контекстная привязка

Иногда у вас может быть два класса, использующих один и тот же интерфейс, но вы хотите внедрить в каждый класс разные реализации. Например, два контроллера могут зависеть от разных реализаций Illuminate\Contracts\Filesystem\Filesystem контракта. Laravel предоставляет простой и удобный интерфейс для определения такого поведения:

```

use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('local');
    });

$this->app->when([VideoController::class, UploadController::class])
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('s3');
    });

```

Рис.44. пример

Связывание примитивов

Иногда у вас может быть класс, который получает некоторые внедренные классы, но также нуждается в внедренном примитивном значении, таком как целое число. Вы можете легко использовать контекстную привязку для ввода любого значения, которое может понадобиться вашему классу:

```

use App\Http\Controllers\UserController;

$this->app->when(UserController::class)
    ->needs('$variableName')
    ->give($value);

```

Рис.45. пример

Иногда класс может зависеть от массива помеченных экземпляров. Используя этот giveTagged метод, вы можете легко внедрить этот тег во все привязки контейнера:

```
$this->app->when(ReportAggregator::class)
    ->needs('$reports')
    ->giveTagged('reports');
```

Рис.46. пример

Если вам нужно ввести значение из одного из файлов конфигурации вашего приложения, вы можете использовать giveConfig метод:

```
$this->app->when(ReportAggregator::class)
    ->needs('$timezone')
    ->giveConfig('app.timezone');
```

Рис.47. пример

Привязка типизированных вариадиков

Иногда у вас может быть класс, который получает массив типизированных объектов с использованием аргумента конструктора с переменным числом аргументов:

```
<?php

use App\Models\Firewall;
use App\Services\Logger;

class Firewall
{
    /**
     * The filter instances.
     *
     * @var array
     */
    protected $filters;

    /**
     * Create a new class instance.
     */
    public function __construct(
        protected Logger $logger,
        Firewall ...$filters,
    ) {
        $this->filters = $filters;
    }
}
```

Рис.48. пример

Используя контекстную привязку, вы можете разрешить эту зависимость, предоставив методу `give` замыкание, которое возвращает массив разрешенных `Filter` экземпляров:

```
$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give(function (Application $app) {
        return [
            $app->make(NullFilter::class),
            $app->make(ProfanityFilter::class),
            $app->make(TooLongFilter::class),
        ];
    });
```

Рис.49. пример

Для удобства вы также можете просто предоставить массив имен классов, который будет разрешаться контейнером всякий раз, когда `Firewall` понадобятся `Filter` экземпляры:

```
$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give([
        NullFilter::class,
        ProfanityFilter::class,
        TooLongFilter::class,
    ]);
```

Рис. 50. пример

Зависимости вариативных тегов

Иногда класс может иметь вариативную зависимость, которая указана как заданный класс (`Report ...$reports`). Используя методы `needs` и, вы можете легко внедрить этот тег `giveTagged` во все привязки контейнера для данной зависимости:

```
$this->app->when(ReportAggregator::class)
    ->needs(Report::class)
    ->giveTagged('reports');
```

Рис.51. пример

Тегирование

Иногда вам может потребоваться разрешить все привязки определенной «категории». Например, возможно, вы создаете анализатор отчетов, который получает массив множества различных Report реализаций интерфейса. После регистрации Report реализаций вы можете присвоить им тег с помощью tag метода:

```
$this->app->bind(CpuReport::class, function () {
    // ...
});

$this->app->bind(MemoryReport::class, function () {
    // ...
});

$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');
```

Рис.52. пример

После того как службы были помечены, вы можете легко разрешить их все с помощью метода контейнера tagged:

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

Рис.53. пример

Расширение привязок

Метод extend позволяет модифицировать разрешенные услуги. Например, когда служба разрешена, вы можете запустить дополнительный код для украшения или настройки службы. Метод extend принимает два аргумента: класс сервиса, который вы расширяете, и замыкание, которое должно возвращать измененный сервис. Замыкание получает разрешаемую службу и экземпляр контейнера:

```
$this->app->extend(Service::class, function (Service $service, Application $app)
    return new DecoratedService($service);
});
```

Рис.54. пример

Разрешение

Метод make__

Вы можете использовать этот make метод для разрешения экземпляра класса из контейнера. Метод make принимает имя класса или интерфейса, который вы хотите разрешить:

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

Рис.55. пример

Если некоторые зависимости вашего класса не разрешимы через контейнер, вы можете внедрить их, передав в метод как ассоциативный массив makeWith. Например, мы можем вручную передать \$id аргумент конструктора, необходимый службе Transistor:

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

Рис.56. пример

Этот bound метод можно использовать для определения того, был ли класс или интерфейс явно привязан к контейнеру:

```
if ($this->app->bound(Transistor::class)) {
    // ...
}
```

Рис.57. пример

Если вы находитесь за пределами поставщика услуг в месте вашего кода, у которого нет доступа к переменной \$app, вы можете использовать App facade или app помощник для разрешения экземпляра класса из контейнера:

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);

$transistor = app(Transistor::class);
```

Рис.58. пример

Если вы хотите, чтобы сам экземпляр контейнера Laravel был внедрен в класс, который разрешается контейнером, вы можете указать класс Illuminate\Container\Container в конструкторе вашего класса:

```
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 */
public function __construct(
    protected Container $container
) {}
```

Рис.59. пример

Автоматическая инъекция

В качестве альтернативы, что важно, вы можете указать зависимость в конструкторе класса, который разрешается контейнером, включая контроллеры, прослушиватели событий, промежуточное программное обеспечение и многое другое. Кроме того, вы можете указать зависимости в handle методе заданий в очереди. На практике именно так контейнер должен разрешать большинство ваших объектов.

Например, вы можете указать репозиторий, определенный вашим приложением, в конструкторе контроллера. Репозиторий будет автоматически разрешен и внедрен в класс:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Show the user with the given ID.
     */
    public function show(string $id): User
    {
        $user = $this->users->findOrFail($id);

        return $user;
    }
}
```

Рис.60. пример

Вызов метода и внедрение

Иногда вам может потребоваться вызвать метод экземпляра объекта, позволяя контейнеру автоматически внедрять зависимости этого метода. Например, учитывая следующий класс:

```

<?php

namespace App;

use App\Repositories\UserRepository;

class UserReport
{
    /**
     * Generate a new user report.
     */
    public function generate(UserRepository $repository): array
    {
        return [
            // ...
        ];
    }
}

```

Рис.61. пример

Вы можете вызвать generate метод через контейнер следующим образом:

```

use App\UserReport;
use Illuminate\Support\Facades\App;

$report = App::call([new UserReport, 'generate']);

```

Рис.62. пример

Этот call метод принимает любой вызов PHP. Метод контейнера call можно даже использовать для вызова замыкания при автоматическом внедрении его зависимостей:

```

use App\Repositories\UserRepository;
use Illuminate\Support\Facades\App;

$result = App::call(function (UserRepository $repository) {
    // ...
});

```

Рис.63. пример

Контейнерные события

Контейнер службы генерирует событие каждый раз, когда он разрешает объект. Вы можете прослушать это событие, используя `resolving` метод:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor, Application $app) {
    // Called when container resolves objects of type "Transistor"...
});

$this->app->resolving(function (mixed $object, Application $app) {
    // Called when container resolves object of any type...
});
```

Рис.64. пример

Как видите, разрешаемый объект будет передан обратному вызову, что позволит вам установить любые дополнительные свойства объекта, прежде чем он будет передан его потребителю.

PSR-11

Сервисный контейнер Laravel реализует интерфейс PSR-11. Поэтому вы можете ввести подсказку интерфейса контейнера PSR-11, чтобы получить экземпляр контейнера Laravel:

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    // ...
});
```

Рис.65. пример

Исключение выдается, если данный идентификатор не может быть разрешен. Исключением будет случай, `Psr\Container\NotFoundExceptionInterface` если идентификатор никогда не был привязан. `Psr\Container\ContainerExceptionInterface` Если

идентификатор был привязан, но его не удалось разрешить, будет создан экземпляр.

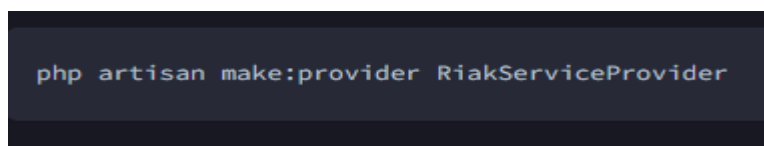
9. ПОСТАВЩИКИ УСЛУГ

Если вы откроете config/app.php файл, включенный в Laravel, вы увидите providers массив. Это все классы поставщика услуг, которые будут загружены для вашего приложения. По умолчанию в этом массиве указан набор основных поставщиков услуг Laravel. Эти провайдеры загружают основные компоненты Laravel, такие как почтовая программа, очередь, кеш и другие.

Поставщики услуг письма

Все поставщики услуг расширяют этот Illuminate\Support\ServiceProvider класс. Большинство поставщиков услуг содержат метод register и boot. Внутри register метода вам следует привязывать вещи только к сервисному контейнеру. Никогда не пытайтесь зарегистрировать в register методе какие-либо прослушиватели событий, маршруты или любые другие функциональные возможности.

Artisan CLI может сгенерировать нового провайдера с помощью make:provider команды:



```
php artisan make:provider RiakServiceProvider
```

Рис.66. пример

Метод регистрации

Как упоминалось ранее, внутри register метода вам следует привязывать вещи только к сервисному контейнеру. Никогда не пытайтесь зарегистрировать в register методе какие-либо прослушиватели событий, маршруты или любые другие функциональные возможности. В противном случае вы можете случайно воспользоваться услугой, предоставляемой поставщиком услуг, которая еще не загружена.

Давайте посмотрим на базового поставщика услуг. В любом из методов вашего поставщика услуг у вас всегда есть доступ к \$app свойству, которое обеспечивает доступ к контейнеру службы:


```

<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection(config('riak'));
        });
    }
}

```

Рис.67. пример

Этот поставщик услуг определяет только register метод и использует этот метод для определения реализации App\Services\Riak\Connection в контейнере службы. Если вы еще не знакомы с сервисным контейнером Laravel, ознакомьтесь с его документацией.

Свойства bindings и singletons__

Если ваш поставщик услуг регистрирует множество простых привязок, вы можете использовать свойства bindings и singletons вместо ручной регистрации каждой привязки контейнера. Когда поставщик услуг загружается платформой, он автоматически проверяет эти свойства и регистрирует их привязки:

```

<?php

namespace App\Providers;

use App\Contracts\DowntimeNotifier;
use App\Contracts\ServerProvider;
use App\Services\DigitalOceanServerProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * All of the container bindings that should be registered.
     *
     * @var array
     */
    public $bindings = [
        ServerProvider::class => DigitalOceanServerProvider::class,
    ];

    /**
     * All of the container singletons that should be registered.
     *
     * @var array
     */
    public $singletons = [
        DowntimeNotifier::class => PingdomDowntimeNotifier::class,
        ServerProvider::class => ServerToolsProvider::class,
    ];
}

```

Рис.68. пример

Метод загрузки

Итак, что, если нам нужно зарегистрировать композитор представления у нашего поставщика услуг? Это должно быть сделано внутри boot метода. Этот метод вызывается после регистрации всех других поставщиков

услуг, что означает, что у вас есть доступ ко всем другим сервисам, зарегистрированным в рамках:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        View::composer('view', function () {
            // ...
        });
    }
}
```

Рис.69. пример

Внедрение зависимостей метода загрузки

Вы можете указать зависимости для boot метода вашего поставщика услуг. Контейнер службы автоматически внедрит все необходимые вам зависимости:

```
use Illuminate\Contracts\Routing\ResponseFactory;

/**
 * Bootstrap any application services.
 */
public function boot(ResponseFactory $response): void
{
    $response->macro('serialized', function (mixed $value) {
        // ...
    });
}
```

Рис.70. пример

Регистрация провайдеров

Все поставщики услуг зарегистрированы в config/app.php файле конфигурации. Этот файл содержит providers массив, в котором вы можете перечислить имена классов ваших поставщиков услуг. По умолчанию в этом массиве зарегистрирован набор основных поставщиков услуг Laravel. Поставщики по умолчанию загружают основные компоненты Laravel, такие как почтовая программа, очередь, кеш и другие.

Чтобы зарегистрировать своего провайдера, добавьте его в массив:

```
'providers' => ServiceProvider::defaultProviders()->merge([
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
])->toArray(),
```

Рис.71. пример

Отложенные поставщики

Если ваш провайдер регистрирует только привязки в сервисном контейнере, вы можете отложить его регистрацию до тех пор, пока одна из зарегистрированных привязок действительно не понадобится. Отсрочка загрузки такого провайдера улучшит производительность вашего приложения, поскольку оно не загружается из файловой системы при каждом запросе.

Laravel компилирует и хранит список всех сервисов, предоставляемых отложенными поставщиками услуг, вместе с именем класса своего поставщика услуг. Затем, только когда вы пытаетесь разрешить одну из этих служб, Laravel загружает поставщика услуг.

Чтобы отложить загрузку поставщика, реализуйте интерфейс \Illuminate\Contracts\Support\DeferrableProvider и определите provides метод. Метод provides должен возвращать привязки сервисного контейнера, зарегистрированные провайдером:

```

<?php

namespace App\Providers;

use App\Services\iak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Contracts\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Get the services provided by the provider.
     *
     * @return array<int, string>
     */
    public function provides(): array
    {
        return [Connection::class];
    }
}

```

Рис.72. пример

10. FACADE

Фасады Laravel служат «статическими прокси» для базовых классов в сервисном контейнере, обеспечивая преимущества краткого и выразительного синтаксиса, сохраняя при этом большую тестируемость и гибкость, чем традиционные статические методы. Если вы не до конца понимаете, как работают фасады, это совершенно нормально — просто плывите по течению и продолжайте изучать Laravel.

Все фасады Laravel определены в пространстве `Illuminate\Support\Facades` имен. Итак, мы можем легко получить доступ к такому фасаду:

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Route;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Рис.73. пример

В документации Laravel во многих примерах будут использоваться фасады для демонстрации различных функций фреймворка.

Вспомогательные функции

В дополнение к фасадам Laravel предлагает множество глобальных «вспомогательных функций», которые еще больше упрощают взаимодействие с распространенными функциями Laravel. Некоторые из общих вспомогательных функций, с которыми вы можете взаимодействовать `view`: `response`, `url`, `config`, и другие. Каждая вспомогательная функция, предлагаемая Laravel, документирована с указанием соответствующей функции; однако полный список доступен в специальной вспомогательной документации.

Например, вместо использования `Illuminate\Support\Facades\Response` фасада для генерации ответа JSON мы можем просто использовать `response` функцию. Поскольку вспомогательные функции доступны глобально, вам не нужно импортировать какие-либо классы, чтобы их использовать:

```

use Illuminate\Support\Facades\Response;

Route::get('/users', function () {
    return Response::json([
        // ...
    ]);
});

Route::get('/users', function () {
    return response()->json([
        // ...
    ]);
});

```

Рис.74. пример

Когда использовать фасады

Фасады имеют множество преимуществ. Они предоставляют краткий, запоминающийся синтаксис, который позволяет вам использовать функции Laravel, не запоминая длинные имена классов, которые необходимо вводить или настраивать вручную. Более того, благодаря уникальному использованию динамических методов PHP их легко тестировать.

Однако при использовании фасадов необходимо соблюдать некоторую осторожность. Основной опасностью фасадов является класс «ползучесть размаха». Поскольку фасады настолько просты в использовании и не требуют внедрения, ваши классы могут продолжать расти и использовать множество фасадов в одном классе. При использовании внедрения зависимостей этот потенциал смягчается визуальной обратной связью, которую большой конструктор дает вам о том, что ваш класс становится слишком большим. Поэтому при использовании фасадов обратите особое внимание на размер вашего класса, чтобы его сфера ответственности оставалась узкой. Если ваш класс становится слишком большим, рассмотрите возможность разделения его на несколько более мелкие классы.

Фасады против. Внедрение зависимости

Одним из основных преимуществ внедрения зависимостей является возможность менять реализации внедренного класса. Это полезно во время тестирования, поскольку вы можете внедрить макет или заглушку и подтвердить, что в заглушке были вызваны различные методы.

Как правило, невозможно имитировать или заглушить действительно статический метод класса. Однако, поскольку фасады используют

динамические методы для прокси-вызовов методов к объектам, разрешенным из сервисного контейнера, мы фактически можем тестировать фасады так же, как мы тестировали бы внедренный экземпляр класса. Например, учитывая следующий маршрут:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Рис.75. пример

Используя методы фасадного тестирования Laravel, мы можем написать следующий тест, чтобы убедиться, что метод `Cache::get` был вызван с ожидаемым аргументом:

```
use Illuminate\Support\Facades\Cache;

/**
 * A basic functional test example.
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

Рис.76. пример

Фасады против. Вспомогательные функции

Помимо фасадов, Laravel включает в себя множество «вспомогательных» функций, которые могут выполнять общие задачи, такие как создание представлений, запуск событий, диспетчеризация заданий или отправка HTTP-ответов. Многие из этих вспомогательных функций выполняют ту же функцию, что и соответствующий фасад. Например, этот вызов фасада и вызов помощника эквивалентны:


```
return Illuminate\Support\Facades\View::make('profile');

return view('profile');
```

Рис.77. пример

Между фасадами и вспомогательными функциями нет абсолютно никакой практической разницы. При использовании вспомогательных функций вы все равно можете тестировать их точно так же, как и соответствующий фасад. Например, учитывая следующий маршрут:

```
Route::get('/cache', function () {
    return cache('key');
});
```

Рис.78. пример

Помощник cache вызовет get метод класса, лежащего в основе Cache фасада. Итак, несмотря на то, что мы используем вспомогательную функцию, мы можем написать следующий тест, чтобы убедиться, что метод был вызван с ожидаемым аргументом:

```
use Illuminate\Support\Facades\Cache;

/**
 * A basic functional test example.
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

Рис.79. пример

Как работают фасады

В приложении Laravel фасад — это класс, обеспечивающий доступ к объекту из контейнера. Машины, которые делают эту работу, находятся в Facade классе. Фасады Laravel и любые созданные вами пользовательские фасады расширяют базовый Illuminate\Support\Facades\Facade класс.

Базовый Facade класс использует __callStatic() магический метод для отсрочки вызовов вашего фасада к объекту, разрешенному из контейнера. В

приведенном ниже примере выполняется вызов системы кэширования Laravel. Взглянув на этот код, можно предположить, что get в классе вызывается статический метод Cache:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function showProfile(string $id): View
    {
        $user = Cache::get('user:'.$id);

        return view('profile', ['user' => $user]);
    }
}
```

Рис.80. пример

Обратите внимание, что в верхней части файла мы «импортируем» Cache фасад. Этот фасад служит прокси для доступа к базовой реализации интерфейса `Illuminate\Contracts\Cache\Factory`. Любые вызовы, которые мы делаем с использованием фасада, будут переданы базовому экземпляру службы кэширования Laravel.

Если мы посмотрим на этот `Illuminate\Support\Facades\Cache` класс, вы увидите, что здесь нет статического метода `get`:

```
class Cache extends Facade
{
    /**
     * Get the registered name of the component.
     */
    protected static function getFacadeAccessor(): string
    {
        return 'cache';
    }
}
```

Рис.81. пример

Вместо этого Cache фасад расширяет базовый Facade класс и определяет метод `getFacadeAccessor()`. Задача этого метода — вернуть имя привязки сервисного контейнера. Когда пользователь ссылается на любой статический метод фасада Cache, Laravel разрешает `cache` привязку из сервисного контейнера и запускает запрошенный метод (в данном случае `get`) для этого объекта.

Фасады реального времени

Используя фасады реального времени, вы можете относиться к любому классу вашего приложения как к фасаду. Чтобы проиллюстрировать, как это можно использовать, давайте сначала рассмотрим код, который не использует фасады реального времени. Например, предположим, что в нашей Podcast модели есть `publish` метод. Однако, чтобы опубликовать подкаст, нам нужно внедрить экземпляр `Publisher`:

```

<?php

namespace App\Models;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     */
    public function publish(Publisher $publisher): void
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
    }
}

```

Рис.82. пример

Внедрение реализации издателя в метод позволяет нам легко протестировать метод изолированно, поскольку мы можем издеваться над внедренным издателем. Однако для этого нам необходимо всегда передавать экземпляр издателя каждый раз, когда мы вызываем publish метод. Используя фасады реального времени, мы можем поддерживать ту же тестируемость, не требуя при этом явной передачи экземпляра Publisher. Чтобы создать фасад реального времени, добавьте к пространству имен импортированного класса префикс Facades:

```

<?php

namespace App\Models;

use App\Contracts\Publisher;
use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     */
    public function publish(): void
    {
        $this->update(['publishing' => now()]);

        Publisher::publish($this);
    }
}

```

Рис.83. пример

При использовании фасада реального времени реализация издателя будет разрешена из контейнера службы с использованием части имени интерфейса или класса, которая появляется после префикса Facades. При тестировании мы можем использовать встроенные помощники по фасадному тестированию Laravel, чтобы имитировать вызов этого метода:

```

<?php

namespace Tests\Feature;

use App\Models\Podcast;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class PodcastTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A test example.
     */
    public function test_podcast_can_be_published(): void
    {
        $podcast = Podcast::factory()->create();

        Publisher::shouldReceive('publish')->once()->with($podcast);

        $podcast->publish();
    }
}

```

Рис.84. пример

Справочник классов фасадов

Ниже вы найдете каждый фасад и его основной класс. Это полезный инструмент для быстрого изучения документации API для данного корня фасада. Ключ привязки сервисного контейнера также включен, где это применимо.

Фасад	Сорт	Привязка сервисного контейнера
Приложение	Освещение\Фундамент\Применение	<code>app</code>
Ремесленник	Illuminate\Contracts\Console\Kernel	<code>artisan</code>
Авторизация	Иллюминат\Аут\Аутманажер	<code>auth</code>
Аутентификация (экземпляр)	Illuminate\Contracts\Auth\Guard	<code>auth.driver</code>
Лезвие	Illuminate\View\Compilers\BladeCompiler	<code>blade.compiler</code>
Транслировать	Освещение\Контракты\Вещание\Фабрика	
Трансляция (экземпляр)	Освещать\Контракты\Вещание\Вещательная компания	
Автобус	Освещать\Контракты\Автобус\Диспетчер	
Кэш	Иллюминат\Кэш\Кэшманагер	<code>cache</code>
Кэш (экземпляр)	Освещать\Кэш\Репозиторий	<code>cache.store</code>
Конфигурация	Освещение\Конфигурация\Репозиторий	<code>config</code>
печенье	Illuminate\Cookie\CookieJar	<code>cookie</code>
Склеп	Подсветка\Шифрование\Шифрование	<code>encrypter</code>
Дата	Illuminate\Support\DateFactory	<code>date</code>
БД	Illuminate\Database\DatabaseManager	<code>db</code>
БД (экземпляр)	Освещение\База данных\Соединение	<code>db.connection</code>
Событие	Освещать\События\Диспетчер	<code>events</code>
Файл	Подсветка\Файловая система\Файловая система	<code>files</code>

Рис.85. справочник

Юрота	Освещать\Контракты\Аутентификация\Доступ\Гейт	
Эш	Illuminate\Contracts\Hashing\Hasher	hash
HTTP	Освещать\Http\Клиент\Фабрика	
Танг	Illuminate\Перевод\Переводчик	translator
Логирование	Illuminate\Log\LogManager	log
Почта	Подсветка\Почта\Почтовая программа	mailer
Уведомление	Illuminate\Notifications\ChannelManager	
Пароль	Illuminate\Auth\Passwords>PasswordBrokerManager	auth.password
Пароль экземпляр)	Illuminate\Auth\Passwords>PasswordBroker	auth.password.broker
Конвейер экземпляр)	Подсветка\Конвейер\Конвейер	
Процесс	Освещать\Процесс\Фабрика	
Очередь	Illuminate\Queue\QueueManager	queue
Очередь экземпляр)	Illuminate\Контракты\Очередь\Очередь	queue.connection
Очередь базовый класс)	Подсветка\Очередь\Очередь	
Перенаправление	Подсветка\Маршрутизация\Перенаправление	redirect
Редис	Иллюминат\Редис\Редисменеджер	redis

Рис.86. справочник

Редис (экземпляр)	Illuminate\Redis\Connections\Connection	redis.connection
Запрос	Освещать\Http\Запрос	request
Ответ	Illuminate\Contracts\Routing\ResponseFactory	
Ответ (экземпляр)	Освещать\Http\Response	
Маршрут	Подсветка\Маршрутизация\Маршрутизатор	router
Схема	Illuminate\Database\Schema\Builder	
Сессия	Illuminate\Session\SessionManager	session
Сеанс (экземпляр)	Освещать\Сессия\Хранить	session.store
Хранилище	Illuminate\Filesystem\FilesystemManager	filesystem
Хранилище (экземпляр)	Освещать\Контракты\Файловая система\Файловая система	filesystem.disk
URL-адрес	Illuminate\Routing\UrlGenerator	url
Валидатор	Освещение\Проверка\Фабрика	validator
Валидатор (экземпляр)	Подсветка\Проверка\Валидатор	
Вид	Освещать\Вид\Фабрика	view
Просмотр (экземпляр)	Освещать\Просмотр\Просмотр	
Вите	Illuminate\Foundation\Vite	

Рис.87. справочник

11.Routing

Базовая маршрутизация

Самые основные маршруты Laravel принимают URI и замыкание, предоставляя очень простой и выразительный метод определения маршрутов и поведения без сложных файлов конфигурации маршрутизации:

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

Рис.88. пример

Файлы маршрутов по умолчанию

Все маршруты Laravel определены в ваших файлах маршрутов, которые расположены в routes каталоге. Эти файлы автоматически загружаются вашим приложением App\Providers\RouteServiceProvider. Этот routes/web.php файл определяет маршруты для вашего веб-интерфейса. Этим маршрутам назначается web группа промежуточного программного обеспечения, которая предоставляет такие функции, как состояние сеанса и защита CSRF. Маршруты routes/api.php не сохраняют состояние и им назначается api группа промежуточного программного обеспечения.

Для большинства приложений вы начнете с определения маршрутов в routes/web.php файле. Доступ к маршрутам, определенным в , routes/web.php можно получить, введя URL-адрес определенного маршрута в браузере. Например, вы можете получить доступ к следующему маршруту, перейдя <http://example.com/user> в браузере:

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);
```

Рис.89. пример

Маршруты, определенные в routes/api.php файле, вложены в группу маршрутов с помощью расширения RouteServiceProvider. Внутри этой группы /api префикс URI применяется автоматически, поэтому вам не нужно вручную применять его к каждому маршруту в файле. Вы можете изменить префикс и другие параметры группы маршрутов, изменив свой RouteServiceProvider класс.

Доступные методы маршрутизатора

Маршрутизатор позволяет прописывать маршруты, отвечающие на любой HTTP-команд:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Рис.90. пример

Иногда вам может потребоваться зарегистрировать маршрут, отвечающий на несколько команд HTTP. Вы можете сделать это, используя match метод. Или вы даже можете зарегистрировать маршрут, который отвечает на все команды HTTP, используя any метод:

```
Route::match(['get', 'post'], '/', function () {
    // ...
});

Route::any('/', function () {
    // ...
});
```

Рис.91. пример

Внедрение зависимости

Вы можете указать любые зависимости, необходимые для вашего маршрута, в сигнатуре обратного вызова вашего маршрута. Объявленные зависимости будут автоматически разрешены и внедрены в обратный вызов сервисным контейнером Laravel . Например, вы можете указать Illuminate\Http\Request классу, чтобы текущий HTTP-запрос автоматически вводился в обратный вызов вашего маршрута:

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

Рис.92. Пример

CSRF-защита

Помните, что любые HTML-формы, указывающие на POST, или маршруты, определенные в файле маршрутов, должны включать поле токена CSRF. В противном случае запрос будет отклонен. Подробнее о защите CSRF можно прочитать в документации CSRF :PUTPATCHDELETEweb

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Рис.93. пример

Перенаправление маршрутов

Если вы определяете маршрут, который перенаправляет на другой URI, вы можете использовать этот Route::redirect метод. Этот метод предоставляет удобный ярлык, позволяющий не определять полный маршрут или контроллер для выполнения простого перенаправления:

```
Route::redirect('/here', '/there');
```

Рис.94. пример

По умолчанию Route::redirect возвращает 302 код состояния. Вы можете настроить код состояния, используя необязательный третий параметр:

```
Route::redirect('/here', '/there', 301);
```

Рис.95. пример

Или вы можете использовать `Route::permanentRedirect` метод для возврата 301 кода состояния:

```
Route::permanentRedirect('/here', '/there');
```

Рис.96. пример

Посмотреть маршруты

Если ваш маршрут должен возвращать только представление, вы можете использовать этот `Route::view` метод. Как и `redirect` метод, этот метод предоставляет простой ярлык, поэтому вам не нужно определять полный маршрут или контроллер. Метод `view` принимает URI в качестве первого аргумента и имя представления в качестве второго аргумента. Кроме того, вы можете предоставить массив данных для передачи представлению в качестве необязательного третьего аргумента:

```
Route::view('/welcome', 'welcome');  
  
Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Рис.97. пример

Список маршрутов

Команда `route:list` Artisan может легко предоставить обзор всех маршрутов, определенных вашим приложением:

```
php artisan route:list
```

Рис.98. пример

По умолчанию промежуточное программное обеспечение маршрута, назначенное каждому маршруту, не будет отображаться в `route:list` выходных данных; однако вы можете поручить Laravel отобразить промежуточное программное обеспечение маршрута, добавив параметр `-v` в команду:

```
php artisan route:list -v
```

Рис.99. пример

Вы также можете указать Laravel показывать только маршруты, начинающиеся с заданного URI:

```
php artisan route:list --path=api
```

Рис.100. пример

Кроме того, вы можете указать Laravel скрывать любые маршруты, определенные сторонними пакетами, предоставив опцию `--except-vendor` при выполнении `route:list` команды:

```
php artisan route:list --except-vendor
```

Рис.101. пример

Аналогичным образом вы также можете указать Laravel показывать только маршруты, определенные сторонними пакетами, предоставив опцию `--only-vendor` при выполнении `route:list` команды:

```
php artisan route:list --only-vendor
```

Рис.102. пример

Обязательные параметры

Иногда вам нужно будет захватить сегменты URI внутри вашего маршрута. Например, вам может потребоваться получить идентификатор пользователя из URL-адреса. Вы можете сделать это, определив параметры маршрута:

```
Route::get('/user/{id}', function (string $id) {  
    return 'User '.$id;  
});
```

Рис.103. пример

Вы можете определить столько параметров маршрута, сколько требуется для вашего маршрута:

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string $postId, string $commentId) {
    // ...
});
```

Рис.104. пример

Параметры маршрута всегда заключаются в {} фигурные скобки и должны состоять из буквенных символов. Символы подчеркивания (_) также допустимы в именах параметров маршрута. Параметры маршрута вводятся в обратные вызовы/контроллеры маршрута в соответствии с их порядком — имена аргументов обратного вызова маршрута/контроллера не имеют значения.

Параметры и внедрение зависимостей

Если у вашего маршрута есть зависимости, которые вы хотите, чтобы сервисный контейнер Laravel автоматически вводил в обратный вызов вашего маршрута, вам следует указать параметры маршрута после зависимостей:

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, string $id) {
    return 'User '.$id;
});
```

Рис.105. пример

Дополнительные параметры

Иногда вам может потребоваться указать параметр маршрута, который не всегда присутствует в URI. Это можно сделать, поставив отметку ? после имени параметра. Обязательно присвойте соответствующей переменной маршрута значение по умолчанию:

```
Route::get('/user/{name?}', function (?string $name = null) {
    return $name;
});

Route::get('/user/{name?}', function (?string $name = 'John') {
    return $name;
});
```

Рис.106. пример

Ограничения регулярных выражений

Вы можете ограничить формат параметров вашего маршрута, используя `where` метод экземпляра маршрута. Метод `where` принимает имя параметра и регулярное выражение, определяющее, как следует ограничить параметр:

```
Route::get('/user/{name}', function (string $name) {
    // ...
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function (string $id) {
    // ...
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Рис.107. пример

Для удобства некоторые часто используемые шаблоны регулярных выражений имеют вспомогательные методы, которые позволяют быстро добавлять ограничения шаблона в ваши маршруты:


```

Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {
    //
})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);

```

Рис.108. пример

Если входящий запрос не соответствует ограничениям шаблона маршрута, будет возвращен HTTP-ответ 404.

Глобальные ограничения

Если вы хотите, чтобы параметр маршрута всегда ограничивался заданным регулярным выражением, вы можете использовать этот `pattern` метод. Вы должны определить эти шаблоны в `boot` методе вашего `App\Providers\RouteServiceProvider` класса:

```
/**
 * Define your route model bindings, pattern filters, etc.
 */
public function boot(): void
{
    Route::pattern('id', '[0-9]+');
}
```

Рис.109. пример

После определения шаблона он автоматически применяется ко всем маршрутам, использующим это имя параметра:

```
Route::get('/user/{id}', function (string $id) {
    // Only executed if {id} is numeric...
});
```

Рис.110. пример

Закодированные косые черты

Компонент маршрутизации Laravel позволяет всем символам, кроме /присутствующих в значениях параметров маршрута. Вы должны явно разрешить /быть частью вашего заполнителя, используя where регулярное выражение условия:

```
Route::get('/search/{search}', function (string $search) {
    return $search;
})->where('search', '.*');
```

Рис.111. пример

Закодированные косые черты поддерживаются только в пределах последнего сегмента маршрута.

Именованные маршруты

Именованные маршруты позволяют удобно создавать URL-адреса или перенаправления для определенных маршрутов. Вы можете указать имя маршрута, связав name метод с определением маршрута:

```
Route::get('/user/profile', function () {  
    // ...  
})->name('profile');
```

Рис.112. пример

Вы также можете указать имена маршрутов для действий контроллера:

```
Route::get(  
    '/user/profile',  
    [UserProfileController::class, 'show']  
)->name('profile');
```

Рис.113. пример

Имена маршрутов всегда должны быть уникальными.

Генерация URL-адресов для именованных маршрутов

После того как вы присвоили имя данному маршруту, вы можете использовать имя маршрута при создании URL-адресов или перенаправлений с помощью Laravel `route` и `redirect` вспомогательных функций:

```
// Generating URLs...  
$url = route('profile');  
  
// Generating Redirects...  
return redirect()->route('profile');  
  
return to_route('profile');
```

Рис.114. пример

Если именованный маршрут определяет параметры, вы можете передать параметры в качестве второго аргумента функции `route`. Указанные параметры будут автоматически вставлены в сгенерированный URL в правильные позиции:

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1]);
```

Рис.115. пример

Если вы передадите дополнительные параметры в массиве, эти пары ключ/значение будут автоматически добавлены в строку запроса сгенерированного URL-адреса:

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

Рис.116. пример

Проверка текущего маршрута

Если вы хотите определить, был ли текущий запрос направлен на заданный именованный маршрут, вы можете использовать этот `named` метод в экземпляре `Route`. Например, вы можете проверить имя текущего маршрута с помощью промежуточного программного обеспечения маршрута:

```

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Handle an incoming request.
 *
 * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

    return $next($request);
}

```

Рис.117. пример

Группы маршрутов

Группы маршрутов позволяют совместно использовать атрибуты маршрута, например промежуточное программное обеспечение, по большому количеству маршрутов без необходимости определять эти атрибуты для каждого отдельного маршрута.

Вложенные группы пытаются разумно «объединить» атрибуты со своей родительской группой. Промежуточное программное обеспечение и where условия объединяются, а имена и префиксы добавляются. Разделители пространства имен и косые черты в префиксах URI добавляются автоматически, где это необходимо.

Промежуточное ПО

Чтобы назначить промежуточное программное обеспечение всем маршрутам внутри группы, вы можете использовать этот middleware метод до определения группы. Промежуточное программное обеспечение выполняется в том порядке, в котором оно указано в массиве:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

Рис.118. пример

Контроллеры

Если группа маршрутов использует один и тот же контроллер, вы можете использовать этот controller метод для определения общего контроллера для всех маршрутов в группе. Затем при определении маршрутов вам нужно указать только метод контроллера, который они вызывают:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

Рис.119. пример

Маршрутизация субдоменов

Группы маршрутов также могут использоваться для управления маршрутизацией поддоменов. Субдоменам могут быть назначены параметры маршрута так же, как URI маршрута, что позволяет вам захватить часть поддомена для использования в вашем маршруте или контроллере. Субдомен можно указать, вызвав domain метод перед определением группы:

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function (string $account, string $id) {
        // ...
    });
});
```

Рис.120. пример

Чтобы гарантировать доступность маршрутов вашего субдомена, вам следует зарегистрировать маршруты субдомена перед регистрацией маршрутов корневого домена. Это не позволит маршрутам корневого домена перезаписывать маршруты поддоменов, которые имеют одинаковый путь URI.

Префиксы маршрутов

Этот `prefix` метод можно использовать для префикса каждого маршрута в группе с заданным URI. Например, вы можете добавить префикс ко всем URI маршрутов в группе `admin`:

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Рис.121. пример

Префиксы имен маршрутов

Этот `name` метод можно использовать для префикса каждого имени маршрута в группе с заданной строкой. Например, вы можете добавить к именам всех маршрутов в группе префикс `admin`. Данная строка предваряется именем маршрута точно так, как она указана, поэтому мы обязательно предоставим завершающий символ в префиксе:

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

Рис.122. пример

Привязка модели маршрута

При внедрении идентификатора модели в маршрут или действие контроллера вы часто будете запрашивать базу данных, чтобы получить модель, соответствующую этому идентификатору. Привязка модели маршрута Laravel предоставляет удобный способ автоматического внедрения экземпляров модели непосредственно в ваши маршруты. Например, вместо того, чтобы вводить идентификатор пользователя, вы можете внедрить весь `User` экземпляр модели, соответствующий данному идентификатору.

Неявное связывание

Laravel автоматически разрешает модели Eloquent, определенные в маршрутах или действиях контроллера, чьи имена переменных с указанием типа соответствуют имени сегмента маршрута. Например:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Рис.123. пример

Поскольку `$user` тип переменной соответствует `App\Models\User` модели Eloquent, а имя переменной соответствует `{user}` сегменту URI, Laravel автоматически внедрит экземпляр модели, идентификатор которого соответствует соответствующему значению из URI запроса. Если соответствующий экземпляр модели не найден в базе данных, автоматически будет сгенерирован HTTP-ответ 404.

Конечно, неявное связывание возможно и при использовании методов контроллера. Еще раз обратите внимание, что `{user}` сегмент URI соответствует `$user` переменной в контроллере, которая содержит `App\Models\User` подсказку типа:

```
use App\Http\Controllers\UserController;
use App\Models\User;

// Route definition...
Route::get('/users/{user}', [UserController::class, 'show']);

// Controller method definition...
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```

Рис.124. пример

Мягкое удаление модели

Обычно неявная привязка модели не позволяет получить обратно удаленные модели. Однако вы можете указать неявной привязке получить эти модели, привязав метод `withTrashed` определению вашего маршрута:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

Рис.125. пример

Настройка ключа

Иногда вам может потребоваться разрешить модели Eloquent, используя столбец, отличный от `id`. Для этого вы можете указать столбец в определении параметра маршрута:

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

Рис.126. пример

Неявная привязка перечисления

В PHP 8.1 появилась поддержка Enums. В дополнение к этой функции Laravel позволяет вам вводить подсказку для Enum со строковой поддержкой в определении вашего маршрута, и Laravel будет вызывать маршрут только в том случае, если этот сегмент маршрута соответствует допустимому значению Enum. В противном случае HTTP-ответ 404 будет возвращен автоматически. Например, учитывая следующее Enum:

```

<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}

```

Рис.127. пример

Вы можете определить маршрут, который будет вызываться только в том случае, если {category} сегмент маршрута равен fruits или people. В противном случае Laravel вернет HTTP-ответ 404:

```

use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});

```

Рис.128. пример

Явное связывание

Чтобы использовать привязку модели, вам не обязательно использовать неявное разрешение модели Laravel, основанное на соглашениях. Вы также можете явно определить, как параметры маршрута соответствуют моделям. Чтобы зарегистрировать явную привязку, используйте метод маршрутизатора `model`, чтобы указать класс для данного параметра. Вы должны определить явные привязки модели в начале метода `boot` вашего `RouteServiceProvider` класса:

```

use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 */
public function boot(): void
{
    Route::model('user', User::class);

    // ...
}

```

Рис.129. пример

Затем определите маршрут, содержащий {user}параметр:

```

use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    // ...
});

```

Рис.130. пример

Резервные маршруты

Используя этот `Route::fallback` метод, вы можете определить маршрут, который будет выполняться, когда ни один другой маршрут не соответствует входящему запросу. Обычно необработанные запросы автоматически отображают страницу «404» через обработчика исключений вашего приложения. Однако, поскольку вы обычно определяете fallback маршрут внутри своего `routes/web.php` файла, все промежуточное программное обеспечение в `web`группе промежуточного программного обеспечения будет применяться к этому маршруту. При необходимости вы можете добавить к этому маршруту дополнительное промежуточное программное обеспечение:

```

Route::fallback(function () {
    // ...
});

```

Рис.131. пример

Ограничение скорости

Определение ограничителей скорости

Laravel включает в себя мощные и настраиваемые службы ограничения скорости, которые вы можете использовать для ограничения объема трафика для данного маршрута или группы маршрутов. Для начала вам следует определить конфигурации ограничителя скорости, соответствующие потребностям вашего приложения.

Обычно ограничители скорости определяются внутри boot метода класса вашего приложения App\Providers\RouteServiceProvider. Фактически, этот класс уже включает определение ограничителя скорости, которое применяется к маршрутам в routes/api.php файле вашего приложения:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Define your route model bindings, pattern filters, and other route configuratic
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
    });

    // ...
}
```

Рис.132. пример

Прикрепление ограничителей скорости к маршрутам

Ограничители скорости могут быть прикреплены к маршрутам или группам маршрутов с помощью throttle промежуточного программного обеспечения. Промежуточное программное обеспечение дроссельной заслонки принимает имя ограничителя скорости, который вы хотите назначить маршруту:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        // ...
    });

    Route::post('/video', function () {
        // ...
    });
});
```

Рис.133.пример

Регулирование с помощью Redis

Обычно throttle промежуточное программное обеспечение сопоставляется с Illuminate\Routing\Middleware\ThrottleRequests классом. Это сопоставление определено в ядре HTTP вашего приложения (App\Http\Kernel). Однако если вы используете Redis в качестве драйвера кэша вашего приложения, вы можете изменить это сопоставление, чтобы использовать этот Illuminate\Routing\Middleware\ThrottleRequestsWithRedis класс. Этот класс более эффективен при управлении ограничением скорости с помощью Redis:

```
'throttle' => \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
```

Рис.134.пример

Подмена метода формы

HTML-формы не поддерживают действия PUT, PATCH или DELETE. Таким образом, при определении PUT, PATCH или DELETE маршрутов, которые вызываются из HTML-формы, вам необходимо добавить _method в форму скрытое поле. Значение, отправленное с _method полем, будет использоваться в качестве метода HTTP-запроса:

```
<form action="/example" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{ csrf_token() }">
</form>
```

Рис.133.пример

Для удобства вы можете использовать @method директиву Blade для генерации _method поля ввода:

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

Рис.134.пример

Доступ к текущему маршруту

Вы можете использовать методы current, currentRouteName и currentRouteAction на Route фасаде для доступа к информации о маршруте, обрабатывающем входящий запрос:

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

Рис.133. пример

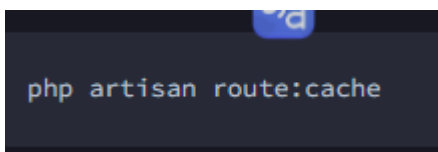
Вы можете обратиться к документации API как для базового класса фасада Route, так и для экземпляра Route, чтобы просмотреть все методы, доступные в классах маршрутизатора и маршрута.

Совместное использование ресурсов между источниками (CORS)

Laravel может автоматически отвечать на OPTIONSHTTP-запросы CORS, используя значения, которые вы настраиваете. Все параметры CORS можно настроить в config/cors.php файле конфигурации вашего приложения. Запросы OPTIONS будут автоматически обрабатываться промежуточным HandleCors программным обеспечением, которое по умолчанию включено в ваш глобальный стек промежуточного программного обеспечения. Глобальный стек промежуточного программного обеспечения находится в HTTP-ядре вашего приложения (App\Http\Kernel).

Кэширование маршрутов

При развертывании вашего приложения в рабочей среде вам следует воспользоваться кэшем маршрутов Laravel. Использование кэша маршрутов значительно сократит время, необходимое для регистрации всех маршрутов вашего приложения. Чтобы сгенерировать кэш маршрутов, выполните route:cache команду Artisan:

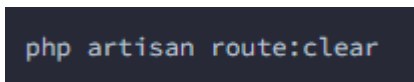


```
php artisan route:cache
```

Рис.134. пример

После запуска этой команды ваш кэшированный файл маршрутов будет загружаться при каждом запросе. Помните: если вы добавите новые маршруты, вам потребуется создать новый кэш маршрутов. По этой причине вам следует запускать route:cache команду только во время развертывания вашего проекта.

Вы можете использовать route:clear команду для очистки кэша маршрутов:



```
php artisan route:clear
```

Рис.135. пример

12.ПРОМЕЖУТОЧНОЕ ПО

Введение

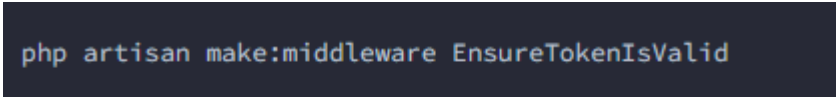
Промежуточное программное обеспечение предоставляет удобный механизм проверки и фильтрации HTTP-запросов, поступающих в ваше приложение. Например, Laravel включает промежуточное программное обеспечение, которое проверяет подлинность пользователя вашего приложения. Если пользователь не аутентифицирован, промежуточное

программное обеспечение перенаправит пользователя на экран входа в ваше приложение. Однако если пользователь аутентифицирован, промежуточное программное обеспечение позволит запросу продолжить работу в приложении.

Дополнительное промежуточное программное обеспечение может быть написано для выполнения множества задач помимо аутентификации. Например, промежуточное программное обеспечение для ведения журналов может регистрировать все входящие запросы к вашему приложению. В структуру Laravel включено несколько промежуточных программ, включая промежуточное программное обеспечение для аутентификации и защиты CSRF. Все это промежуточное программное обеспечение находится в `app/Http/Middleware` каталоге

Определение промежуточного программного обеспечения

Чтобы создать новое промежуточное программное обеспечение, используйте `make:middleware` команду Artisan:



```
php artisan make:middleware EnsureTokenIsValid
```

Рис.136. пример

Эта команда поместит новый `EnsureTokenIsValid` класс в ваш `app/Http/Middleware` каталог. В этом промежуточном программном обеспечении мы разрешаем доступ к маршруту только в том случае, если предоставленные `token` входные данные соответствуют указанному значению. В противном случае мы перенаправим пользователей обратно на `homeURI`:


```

<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

        return $next($request);
    }
}

```

Рис.137.пример

Как видите, если данное значение `token` не соответствует нашему секретному токenu, промежуточное программное обеспечение вернет клиенту HTTP-перенаправление; в противном случае запрос будет передан дальше в приложение. Чтобы передать запрос глубже в приложение (чтобы позволить промежуточному программному обеспечению «пройти»), вам следует вызвать обратный `$next` вызов с помощью метода `$request`.

Лучше всего представлять себе промежуточное программное обеспечение как серию «уровней», через которые HTTP-запросы должны пройти, прежде чем они попадут в ваше приложение. Каждый уровень может изучить запрос и даже полностью отклонить его.

Все промежуточное программное обеспечение разрешается через сервисный контейнер, поэтому вы можете указать любые зависимости, которые вам нужны, в конструкторе промежуточного программного обеспечения.

Промежуточное ПО и ответы

Конечно, промежуточное ПО может выполнять задачи до или после передачи запроса глубже в приложение. Например, следующее промежуточное программное обеспечение выполнит некоторую задачу до того, как запрос будет обработан приложением:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // Perform action

        return $next($request);
    }
}
```

Рис.138.пример

Однако это промежуточное ПО выполнит свою задачу после того, как запрос будет обработан приложением:

```

<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}

```

Рис.139.пример

РЕГИСТРАЦИЯ ПРОМЕЖУТОЧНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Глобальное промежуточное программное обеспечение

Если вы хотите, чтобы промежуточное программное обеспечение запускалось при каждом HTTP-запросе к вашему приложению, укажите класс промежуточного программного обеспечения в свойстве \$middleware вашего app/Http/Kernel.php класса.

Назначение промежуточного программного обеспечения маршрутам

Если вы хотите назначить промежуточное программное обеспечение для определенных маршрутов, вы можете вызвать этот middleware метод при определении маршрута:

```

use App\Http\Middleware\Authenticate;

Route::get('/profile', function () {
    // ...
})->middleware(Authenticate::class);

```

Рис.140.пример

Вы можете назначить маршруту несколько промежуточных программ, передав методу массив имен промежуточных программ middleware:

```
Route::get('/', function () {  
    // ...  
})->middleware([First::class, Second::class]);
```

Рис.141.пример

Для удобства вы можете назначить псевдонимы промежуточному программному обеспечению в файле вашего приложения app/Http/Kernel.php. По умолчанию \$middlewareAliases свойство этого класса содержит записи для промежуточного программного обеспечения, включенного в Laravel. Вы можете добавить в этот список свое собственное промежуточное программное обеспечение и назначить ему псевдоним по вашему выбору:

```
// Within App\Http\Kernel class...  
  
protected $middlewareAliases = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,  
];
```

Рис.142.пример

После того как псевдоним промежуточного программного обеспечения определен в ядре HTTP, вы можете использовать его при назначении промежуточного программного обеспечения маршрутам:

```
Route::get('/profile', function () {  
    // ...  
})->middleware('auth');
```

Рис.143.пример

Исключение промежуточного программного обеспечения

При назначении промежуточного программного обеспечения группе маршрутов иногда может потребоваться запретить применение промежуточного программного обеспечения к отдельному маршруту в группе. Вы можете сделать это, используя `withoutMiddleware` метод:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::middleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/', function () {
        // ...
    });

    Route::get('/profile', function () {
        // ...
    })->withoutMiddleware([EnsureTokenIsValid::class]);
});
```

Рис.144.пример

Вы также можете исключить определенный набор промежуточного программного обеспечения из всей группы определений маршрутов:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

Рис.145.пример

Этот `withoutMiddleware` метод позволяет удалить только промежуточное программное обеспечение маршрута и не применяется к глобальному промежуточному программному обеспечению.

Группы промежуточного программного обеспечения

Иногда вам может потребоваться сгруппировать несколько промежуточных программ под одним ключом, чтобы их было легче назначать маршрутам. Вы можете сделать это, используя `$middlewareGroups` свойства вашего ядра HTTP.

Laravel включает предопределенные группы веб-апи группы промежуточного программного обеспечения, которые содержат общее промежуточное программное обеспечение, которое вы, возможно, захотите применить к своим веб-маршрутам и маршрутам API. Помните, что эти группы промежуточного программного обеспечения автоматически применяются App\Providers\RouteServiceProvider поставщиком услуг вашего приложения к маршрутам в соответствующих файлах веб-апи файлах маршрутов:

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

Рис.146.пример

Группы промежуточного программного обеспечения могут быть назначены маршрутам и действиям контроллера, используя тот же синтаксис, что и индивидуальное промежуточное программное обеспечение. Опять же, группы промежуточного программного обеспечения упрощают одновременное назначение нескольких промежуточных программ одному маршруту:

```
Route::get('/', function () {
    // ...
})->middleware('web');

Route::middleware(['web'])->group(function () {
    // ...
});
```

Рис.147.пример

По умолчанию группы промежуточного программного обеспечения web и api автоматически применяются к соответствующим файлам вашего приложения routes/web.php и routes/api.php файлам App\Providers\RouteServiceProvider.

Сортировка промежуточного программного обеспечения

В редких случаях вам может потребоваться, чтобы ваше промежуточное программное обеспечение выполнялось в определенном порядке, но вы не можете контролировать их порядок, когда они назначаются маршруту. В этом случае вы можете указать приоритет промежуточного программного обеспечения, используя \$middlewarePriority свойство вашего app/Http/Kernel.php файла. Это свойство может отсутствовать в вашем ядре HTTP по умолчанию. Если он не существует, вы можете скопировать его определение по умолчанию ниже:

```
/**
 * The priority-sorted list of middleware.
 *
 * This forces non-global middleware to always be in the given order.
 *
 * @var string[]
 */
protected $middlewarePriority = [
    \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
    \Illuminate\Cookie\Middleware\EncryptCookies::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
    \Illuminate\Contracts\Session\Middleware\AuthenticatesSessions::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    \Illuminate\Auth\Middleware\Authorize::class,
];
```

Рис.148.пример

ПАРАМЕТРЫ ПРОМЕЖУТОЧНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Промежуточное программное обеспечение также может получать дополнительные параметры. Например, если вашему приложению необходимо убедиться, что аутентифицированный пользователь имеет заданную «роль», прежде чем выполнять определенное действие, вы можете создать промежуточное программное обеспечение, `EnsureUserHasRole` которое получает имя роли в качестве дополнительного аргумента.

Дополнительные параметры промежуточного программного обеспечения будут переданы в промежуточное программное обеспечение после `$next` аргумента:

```
1?php
2
3namespace App\Http\Middleware;
4
5use Closure;
6use Illuminate\Http\Request;
7use Symfony\Component\HttpFoundation\Response;
8
9class EnsureUserHasRole
10{
11    /**
12     * Handle an incoming request.
13     *
14     * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
15     */
16    public function handle(Request $request, Closure $next, string $role): Response {
17        {
18            if (! $request->user()->hasRole($role)) {
19                // Redirect...
20            }
21
22            return $next($request);
23        }
24    }
25}
```

Рис.149.пример

Параметры промежуточного программного обеспечения можно указать при определении маршрута, разделив имя и параметры промежуточного программного обеспечения расширением: Несколько параметров должны быть разделены запятыми:

```
Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware('role:editor');
```

Рис.150.пример

ЗАВЕРШАЕМОЕ ПРОМЕЖУТОЧНОЕ ПО

Иногда промежуточному программному обеспечению может потребоваться выполнить некоторую работу после отправки HTTP-ответа в браузер. Если вы определяете `terminate` метод в своем промежуточном программном обеспечении и ваш веб-сервер использует FastCGI, `terminate` метод будет автоматически вызываться после отправки ответа в браузер:

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class TerminatingMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)
     */
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);
    }

    /**
     * Handle tasks after the response has been sent to the browser.
     */
    public function terminate(Request $request, Response $response): void
    {
        // ...
    }
}
```

Рис.151.пример

Метод `terminate` должен получить как запрос, так и ответ. После того как вы определили завершаемое промежуточное программное обеспечение, вам следует добавить его в список маршрутов или глобального промежуточного программного обеспечения в файле `app/Http/Kernel.php`.

При вызове `terminate` метода вашего промежуточного программного обеспечения Laravel разрешит новый экземпляр промежуточного программного обеспечения из сервисного контейнера. Если вы хотите использовать один и тот же экземпляр промежуточного программного обеспечения при вызове методов `handle` и `terminate`, зарегистрируйте промежуточное программное обеспечение в контейнере, используя метод контейнера `singleton`. Обычно это должно быть сделано в `register` методе вашего `AppServiceProvider`:

```
use App\Http\Middleware\TerminatingMiddleware;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(TerminatingMiddleware::class);
}
```

Рис.152.пример

13.ПРЕДОТВРАЩЕНИЕ ЗАПРОСОВ CSRF

Laravel автоматически генерирует «токен» CSRF для каждого активного пользовательского сеанса, управляемого приложением. Этот токен используется для проверки того, что аутентифицированный пользователь действительно отправляет запросы к приложению. Поскольку этот токен хранится в сеансе пользователя и изменяется при каждой регенерации сеанса, вредоносное приложение не может получить к нему доступ.

Доступ к CSRF-токену текущего сеанса можно получить через сеанс запроса или через `csrf_token` вспомогательную функцию:

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

Рис.153.пример

Каждый раз, когда вы определяете HTML-форму «POST», «PUT», «PATCH» или «DELETE» в своем приложении, вам следует включать в форму скрытое поле CSRF, чтобы промежуточное программное обеспечение защиты CSRF могло _token проверить запрос. Для удобства вы можете использовать @csrf директиву Blade для создания поля ввода, скрытого токена:

Исключение URI из защиты CSRF

Иногда вам может потребоваться исключить набор URI из защиты CSRF. Например, если вы используете Stripe для обработки платежей и используете их систему веб-перехватчиков, вам необходимо исключить маршрут обработчика веб-перехватчика Stripe из защиты CSRF, поскольку Stripe не будет знать, какой токен CSRF отправлять на ваши маршруты.

Обычно такие маршруты следует размещать за пределами web группы промежуточного программного обеспечения, которая App\Providers\RouteServiceProvider применяется ко всем маршрутам в routes/web.php файле. Однако вы также можете исключить маршруты, добавив их URI в свойство \$except промежуточного VerifyCsrfToken программного обеспечения:

```
<?php

namespace App\Http\Middleware;

use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ];
}
```

Рис.156.пример

14.КОНТРОЛЛЕРЫ

Контроллеры могут группировать связанную логику обработки запросов в один класс. Например, UserController класс может обрабатывать все входящие запросы, связанные с пользователями, включая отображение, создание, обновление и удаление пользователей. По умолчанию контроллеры хранятся в app/Http/Controllers каталоге.

Написание контроллеров Базовые контроллеры

Чтобы быстро создать новый контроллер, вы можете запустить make:controller команду Artisan. По умолчанию все контроллеры вашего приложения хранятся в app/Http/Controllers каталоге:

```
php artisan make:controller UserController
```

Рис.157.пример

Давайте посмотрим на пример базового контроллера. Контроллер может иметь любое количество общедоступных методов, которые будут отвечать на входящие HTTP-запросы:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Рис.158. пример

После того, как вы написали класс и метод контроллера, вы можете определить маршрут к методу контроллера следующим образом:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Рис.159.пример

Когда входящий запрос соответствует указанному URI маршрута, будет вызван show метод класса, и ему будут переданы параметры маршрута .App\Http\Controllers\UserController

Контроллеры не обязаны расширять базовый класс. Однако у вас не будет доступа к удобным функциям, таким как методы middleware и authorize.

Контроллеры одиночного действия

Если действие контроллера особенно сложное, возможно, вам будет удобно посвятить этому единственному действию целый класс контроллера. Для этого вы можете определить один __invoke метод внутри контроллера:

```
<?php

namespace App\Http\Controllers;

class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     */
    public function __invoke()
    {
        // ...
    }
}
```

Рис.160.пример

При регистрации маршрутов для контроллеров одиночного действия вам не нужно указывать метод контроллера. Вместо этого вы можете просто передать имя контроллера маршрутизатору:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

Рис.161.пример

Вы можете создать вызываемый контроллер, используя опцию `--invokable` команды `make:controller` Artisan:

```
php artisan make:controller ProvisionServer --invokable
```

Рис.162.пример

Промежуточное программное обеспечение контроллера

```
Route::get('profile', [UserController::class, 'show'])->middleware('auth');
```

Рис.163.пример

Или вам может быть удобно указать промежуточное программное обеспечение в конструкторе вашего контроллера. Используя `middleware` метод в конструкторе вашего контроллера, вы можете назначить промежуточное программное обеспечение для действий контроллера:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

Рис.164.пример

Контроллеры также позволяют регистрировать промежуточное ПО с помощью замыкания. Это обеспечивает удобный способ определения встроенного промежуточного программного обеспечения для одного

контроллера без определения всего класса промежуточного программного обеспечения:

```
use Closure;
use Illuminate\Http\Request;

$this->middleware(function (Request $request, Closure $next) {
    return $next($request);
});
```

Рис.165. пример

Контроллеры ресурсов

Если вы думаете о каждой модели Eloquent в своем приложении как о «ресурсе», то обычно для каждого ресурса в вашем приложении выполняются одни и те же наборы действий. Например, представьте, что ваше приложение содержит Photo модель и Movie модель. Вполне вероятно, что пользователи смогут создавать, читать, обновлять или удалять эти ресурсы.

Из-за этого распространенного варианта использования маршрутизация ресурсов Laravel назначает типичные маршруты создания, чтения, обновления и удаления («CRUD») контроллеру с помощью одной строки кода. Для начала мы можем использовать опцию make:controller команды Artisan --resource, чтобы быстро создать контроллер для выполнения этих действий:

```
php artisan make:controller PhotoController --resource
```

Рис.166.пример

Эта команда сгенерирует контроллер по адресу app/Http/Controllers/PhotoController.php. Контроллер будет содержать метод для каждой из доступных операций с ресурсами. Далее вы можете зарегистрировать маршрут ресурса, указывающий на контроллер:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Рис.167.пример

Это объявление одного маршрута создает несколько маршрутов для обработки различных действий с ресурсом. В сгенерированном контроллере уже будут заглушены методы для каждого из этих действий. Помните, что вы всегда можете получить краткий обзор маршрутов вашего приложения, запустив `route:list` команду Artisan.

Вы даже можете зарегистрировать несколько контроллеров ресурсов одновременно, передав методу массив `resources`:

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

Рис.168.пример

Действия, обрабатываемые контроллером ресурсы

Глагол	URI	Действие	Название маршрута
ПОЛУЧАТЬ	<code>/photos</code>	индекс	<code>photos.index</code>
ПОЛУЧАТЬ	<code>/photos/create</code>	создавать	<code>фотографии.создать</code>
ПОЧТА	<code>/photos</code>	магазин	<code>фотографии.магазин</code>
ПОЛУЧАТЬ	<code>/photos/{photo}</code>	показывать	<code>фотографии.показать</code>
ПОЛУЧАТЬ	<code>/photos/{photo}/edit</code>	редактировать	<code>фотографии.редактировать</code>
ПОСТАВИТЬ/ ИСПРАВИТЬ	<code>/photos/{photo}</code>	обновлять	<code>фотографии.обновление</code>
УДАЛИТЬ	<code>/photos/{photo}</code>	разрушать	<code>фотографии.уничтожить</code>

Рис.169.пример

Настройка поведения отсевающей модели

Обычно HTTP-ответ 404 генерируется, если неявно связанная модель ресурса не найдена. Однако вы можете настроить это поведение, вызвав `missing` метод при определении маршрута ресурса. Метод `missing` принимает замыкание, которое будет вызвано, если неявно связанная модель не может быть найдена ни для одного из маршрутов ресурса:


```

use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
    ->missing(function (Request $request) {
        return Redirect::route('photos.index');
    });

```

Рис.170. пример

Мягко удаленные модели

Обычно неявная привязка модели не извлекает модели, которые были обратимо удалены, а вместо этого возвращает HTTP-ответ 404. Однако вы можете указать платформе разрешить обратимое удаление моделей, вызвав метод `withTrashed` при определении маршрута ресурса:

```

use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->withTrashed();

```

Рис.171.пример

Вызов `withTrashed` без аргументов позволит мягко удалять модели для маршрутов `show`, `edit` и `update` ресурсов. Вы можете указать подмножество этих маршрутов, передав методу массив `withTrashed`:

```

Route::resource('photos', PhotoController::class)->withTrashed(['show']);

```

Рис.172.пример

Указание модели ресурсов

Если вы используете привязку модели маршрута и хотите, чтобы методы контроллера ресурсов давали подсказку по типу экземпляра модели, вы можете использовать эту `--model` опцию при создании контроллера:

```

php artisan make:controller PhotoController --model=Photo --resource

```

Рис.173.пример

Генерация запросов формы

Вы можете предоставить `--requests` возможность при создании контроллера ресурсов поручить Artisan сгенерировать классы запроса формы для методов хранения и обновления контроллера:

```
php artisan make:controller PhotoController --model=Photo --resource --requests
```

Рис.174.пример

Частичные маршруты ресурсов

При объявлении маршрута ресурса вы можете указать подмножество действий, которые должен обрабатывать контроллер вместо полного набора действий по умолчанию:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

Рис.175.пример

Маршруты ресурсов API

При объявлении маршрутов ресурсов, которые будут использоваться API, обычно требуется исключить маршруты, представляющие шаблоны HTML, такие как `create` и `edit`. Для удобства вы можете использовать `apiResource` метод для автоматического исключения этих двух маршрутов:

```
use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);
```

Рис.176.пример

Вы можете зарегистрировать сразу несколько контроллеров ресурсов API, передав методу массив `apiResources`:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\PostController;

Route::apiResources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

Рис.177.пример

Чтобы быстро создать контроллер ресурсов API, не включающий методы `create` или `edit`, используйте `--api` переключатель при выполнении `make:controller` команды:

```
php artisan make:controller PhotoController --api
```

Рис.178.пример

Вложенные ресурсы

Иногда вам может потребоваться определить маршруты к вложенному ресурсу. Например, фоторесурс может иметь несколько комментариев, прикрепленных к фотографии. Чтобы вложить контроллеры ресурсов, вы можете использовать «точечную» нотацию в объявлении маршрута:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);
```

Рис.179.пример

Этот маршрут регистрирует вложенный ресурс, доступ к которому можно получить с помощью URI, например:

```
/photos/{photo}/comments/{comment}
```

Рис.180.пример

Определение вложенных ресурсов

Функция неявной привязки модели Laravel может автоматически определять область вложенных привязок, так что разрешенная дочерняя модель подтверждается принадлежностью родительской модели. Используя этот `scoped` метод при определении вложенного ресурса, вы можете включить автоматическую область видимости, а также указать Laravel, по какому полю должен быть получен дочерний ресурс. Дополнительные сведения о том, как это сделать, см. в документации по определению маршрутов ресурсов.

Мелкое вложение

Часто нет необходимости иметь в URI идентификаторы родительского и дочернего элементов, поскольку дочерний идентификатор уже является уникальным идентификатором. При использовании уникальных идентификаторов, таких как автоматически увеличивающиеся первичные ключи, для идентификации ваших моделей в сегментах URI, вы можете выбрать «мелкую вложенность»:

```
use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();
```

Рис.181.пример

Это определение маршрута будет определять следующие маршруты:

Глагол	URI	Действие	Название маршрута
ПОЛУЧАТЬ	/photos/{photo}/comments	индекс	photos.comments.index
ПОЛУЧАТЬ	/photos/{photo}/comments/create	создавать	photos.comments.create
ПОЧТА	/photos/{photo}/comments	магазин	фотографии.комментарии.магазин
ПОЛУЧАТЬ	/comments/{comment}	показывать	комментарии.показать
ПОЛУЧАТЬ	/comments/{comment}/edit	редактировать	комментарии.редактировать
ПОСТАВИТЬ/ ИСПРАВИТЬ	/comments/{comment}	обновлять	комментарии.обновление
УДАЛИТЬ	/comments/{comment}	разрушать	комментарии.уничтожить

Рис.182.пример

Именованние маршрутов ресурсов

По умолчанию все действия контроллера ресурсов имеют имя маршрута; однако вы можете переопределить эти имена, передав массив `names` с желаемыми именами маршрутов:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

Рис.183.пример

Именованние параметров маршрута ресурса

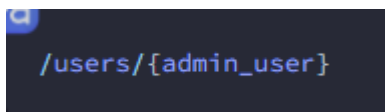
По умолчанию `Route::resource` параметры маршрута для ваших маршрутов ресурсов будут созданы на основе «единственной» версии имени ресурса. Вы можете легко переопределить это для каждого ресурса, используя этот `parameters` метод. Массив, передаваемый в `parameters` метод, должен представлять собой ассоциативный массив имен ресурсов и имен параметров:

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

Рис.184.пример

В приведенном выше примере создается следующий URI для маршрута ресурса `show`:



```
/users/{admin_user}
```

Рис.185.пример

Определение маршрутов ресурсов

Функция неявной привязки модели Laravel с ограниченной областью действия может автоматически определять область вложенных привязок, так что разрешенная дочерняя модель подтверждается как принадлежащая родительской модели. Используя этот `scoped` метод при определении вложенного ресурса, вы можете включить автоматическую область видимости, а также указать Laravel, по какому полю должен быть получен дочерний ресурс:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

Рис.186.пример

Этот маршрут регистрирует вложенный ресурс с ограниченной областью действия, доступ к которому можно получить с помощью URI, например:

```
/photos/{photo}/comments/{comment:slug}
```

Рис.187.пример

При использовании неявной привязки с пользовательским ключом в качестве параметра вложенного маршрута Laravel автоматически определяет область запроса для получения вложенной модели по ее родительскому элементу, используя соглашения для угадывания имени связи в родительском элементе. В этом случае предполагается, что Photoмодель имеет именованную связь `comments` (множественное число имени параметра маршрута), которую можно использовать для извлечения модели `Comment`.

Локализация URL ресурсов

По умолчанию `Route::resource` будут созданы URI ресурсов с использованием английских глаголов и правил множественного числа. Если вам нужно локализовать глаголы действия `create` и `edit`, вы можете использовать этот `Route::resourceVerbs` метод. Это можно сделать в начале метода `boot` вашего приложения `App\Providers\RouteServiceProvider`:

```

/**
 * Define your route model bindings, pattern filters, etc.
 */
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);

    // ...
}

```

Рис.188.пример

Множественный преобразователь Laravel поддерживает несколько разных языков, которые вы можете настроить в соответствии со своими потребностями. После настройки глаголов и языка множественного числа регистрация маршрута ресурса, например, `Route::resource('publicacion', PublicacionController::class)` приведет к созданию следующих URI:

```

/publicacion/crear

/publicacion/{publicaciones}/editar

```

Рис.189.пример

Дополнение контроллеров ресурсов

Если вам нужно добавить дополнительные маршруты к контроллеру ресурсов помимо набора маршрутов ресурсов по умолчанию, вам следует определить эти маршруты перед вызовом метода `Route::resource`; в противном случае маршруты, определенные методом, `resource` могут непреднамеренно иметь приоритет над вашими дополнительными маршрутами:

```
use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

Рис.190.пример

Контроллеры ресурсов Singleton

Иногда ваше приложение будет иметь ресурсы, которые могут иметь только один экземпляр. Например, «профиль» пользователя можно редактировать или обновлять, но у пользователя не может быть более одного «профиля». Аналогично, изображение может иметь одну «миниатюру». Эти ресурсы называются «одноэлементными ресурсами», что означает, что может существовать один и только один экземпляр ресурса. В этих сценариях вы можете зарегистрировать контроллер ресурсов «singleton»:

```
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::singleton('profile', ProfileController::class);
```

Рис.191.пример

Приведенное выше определение одноэлементного ресурса будет регистрировать следующие маршруты. Как видите, маршруты «создания» не регистрируются для одноэлементных ресурсов, а зарегистрированные маршруты не принимают идентификатор, поскольку может существовать только один экземпляр ресурса:

Глагол	URI	Действие	Название маршрута
ПОЛУЧАТЬ	/profile	показывать	профиль.показать
ПОЛУЧАТЬ	/profile/edit	редактировать	профиль.редактировать
ПОСТАВИТЬ/ИСПРАВИТЬ	/profile	обновлять	профиль.обновление

Рис.192.пример

Ресурсы Singleton также могут быть вложены в стандартный ресурс:

В этом примере photos ресурс получит все стандартные маршруты ресурсов; однако thumbnail ресурс будет одноэлементным со следующими маршрутами:

Глагол	URI	Действие	Название маршрута
ПОЛУЧАТЬ	/photos/{photo}/thumbnail	показывать	фотографии.thumbnail.show
ПОЛУЧАТЬ	/photos/{photo}/thumbnail/edit	редактировать	фотографии.thumbnail.edit
ПОСТАВИТЬ/ ИСПРАВИТЬ	/photos/{photo}/thumbnail	обновлять	photos.thumbnail.update

Рис.193.пример

Создаваемые одноэлементным ресурсы

Иногда вам может потребоваться определить маршруты создания и хранения для одноэлементного ресурса. Для этого вы можете вызвать creatable метод при регистрации маршрута одноэлементного ресурса:

```
Route::singleton('photos.thumbnail', ThumbnailController::class)->creatable();
```

Рис.194.пример

В этом примере будут зарегистрированы следующие маршруты. Как видите, DELETE маршрут также будет зарегистрирован для создаваемых одноэлементных ресурсов:

Глагол	URI	Действие	Название маршрута
ПОЛУЧАТЬ	/photos/{photo}/thumbnail/create	создавать	фотографии.thumbnail.create
ПОЧТА	/photos/{photo}/thumbnail	магазин	photos.thumbnail.store
ПОЛУЧАТЬ	/photos/{photo}/thumbnail	показывать	фотографии.thumbnail.show
ПОЛУЧАТЬ	/photos/{photo}/thumbnail/edit	редактировать	фотографии.thumbnail.edit
ПОСТАВИТЬ/ ИСПРАВИТЬ	/photos/{photo}/thumbnail	обновлять	photos.thumbnail.update
УДАЛИТЬ	/photos/{photo}/thumbnail	разрушать	фотографии.thumbnail.destroy

Рис.195.пример

Если вы хотите, чтобы Laravel зарегистрировал DELETE маршрут для одноэлементного ресурса, но не регистрировал маршруты создания или хранения, вы можете использовать метод `destroyable()`:

```
Route::singleton(...)->destroyable();
```

Рис.196.пример

Ресурсы API Singleton

Этот `apiSingleton` метод можно использовать для регистрации одноэлементного ресурса, которым будет управляться через API, что делает ненужными маршруты `create` и `:edit`

```
Route::apiSingleton('profile', ProfileController::class);
```

Рис.197.пример

Конечно, одноэлементные ресурсы API также могут быть `creatable`, которые будут регистрировать `store` и `destroy` маршрутизировать ресурс:

```
Route::apiSingleton('photos.thumbnail', ProfileController::class)->creatable();
```

Рис.198.пример

Внедрение зависимостей контроллеры

Внедрение конструктор

Сервисный контейнер Laravel используется для разрешения всех контроллеров Laravel. В результате вы можете указать любые зависимости, которые могут понадобиться вашему контроллеру, в его конструкторе. Объявленные зависимости будут автоматически разрешены и внедрены в экземпляр контроллера:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * Create a new controller instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}
}
```

Рис.200.пример

Внедрение метода

В дополнение к внедрению в конструктор вы также можете указывать зависимости от методов вашего контроллера. Распространенным вариантом использования внедрения метода является внедрение экземпляра `Illuminate\Http\Request` в методы вашего контроллера:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        // Store the user...

        return redirect('/users');
    }
}
```

Рис.201.пример

Если ваш метод контроллера также ожидает ввода от параметра маршрута, укажите аргументы маршрута после других зависимостей. Например, если ваш маршрут определен так:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

Рис.202.пример

Вы по-прежнему можете ввести подсказку `Illuminate\Http\Request` и получить доступ к своему `id` параметру, определив метод контроллера следующим образом:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the given user.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Update the user...

        return redirect('/users');
    }
}
```

Рис.203.пример