**Определение текущей среды**

Текущая среда приложения определяется с помощью переменной APP\_ENV из вашего файла .env. Вы можете получить доступ к этому значению с помощью метода среды на фасаде приложения:

$environment = App::environment();

Вы также можете передать аргументы методу среды, чтобы проверить, соответствует ли среда заданному значению. Метод вернет истину, если среда соответствует любому из заданных значений:

if (App::environment('local')) {

// Среда является локальной

}

if (App::environment(['local', 'staging'])) {

// Среда либо локальная, либо промежуточная ...

}

**Скрытие переменных среды со страниц отладки**

Когда исключение не перехвачено и переменная среды APP\_DEBUG имеет значение true, на странице отладки будут показаны все переменные среды и их содержимое. В некоторых случаях вам может потребоваться скрыть определенные переменные. Вы можете сделать это, обновив параметр debug\_hide в файле конфигурации config / app.php.

Некоторые переменные доступны как в переменных среды, так и в данных сервера / запроса. Следовательно, вам может потребоваться скрыть их как для $ \_ENV, так и для $ \_SERVER:

return [

// ...

'debug\_hide' => [

'\_ENV' => [

'APP\_KEY',

'DB\_PASSWORD',

],

'\_SERVER' => [

'APP\_KEY',

'DB\_PASSWORD',

],

'\_POST' => [

'password',

],

],

];

**Доступ к значениям конфигурации**

Вы можете легко получить доступ к своим значениям конфигурации с помощью глобальной вспомогательной функции конфигурации 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 может быть указано значение по умолчанию, которое будет возвращено, если параметр конфигурации не существует:

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

// Получить значение по умолчанию, если значение конфигурации не существует ...

$value = config('app.timezone', 'Asia/Seoul');

Чтобы установить значения конфигурации во время выполнения, передайте массив помощнику по настройке:

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

**Кэширование конфигурации**

Чтобы повысить скорость своего приложения, вы должны кэшировать все файлы конфигурации в один файл с помощью Artisan-команды config: cache. Это объединит все параметры конфигурации вашего приложения в один файл, который будет быстро загружен платформой.

Обычно вы должны запускать команду php artisan config: cache как часть процедуры развертывания в производственной среде. Команду не следует запускать во время локальной разработки, поскольку параметры конфигурации часто нужно будет изменять в ходе разработки вашего приложения.

**Режим технического обслуживания**

Когда ваше приложение находится в режиме обслуживания, для всех запросов к вашему приложению будет отображаться настраиваемое представление. Это позволяет легко «отключить» ваше приложение во время его обновления или во время обслуживания. Проверка режима обслуживания включена в стек промежуточного программного обеспечения по умолчанию для вашего приложения. Если приложение находится в режиме обслуживания, будет выброшено исключение MaintenanceModeException с кодом состояния 503.

Чтобы включить режим обслуживания, выполните Artisan-команду down:

php artisan down

Вы также можете указать опцию retry для команды down, которая будет установлена как значение HTTP-заголовка Retry-After:

php artisan down --retry=60

Больше в документации [https://laravel.com/docs/8.x/configuration#maintenance-mode](https://laravel.com/docs/8.x/configuration" \l "maintenance-mode)

**Структура каталогов**

Каталог app содержит основной код вашего приложения. Вскоре мы рассмотрим этот каталог более подробно; однако почти все классы в вашем приложении будут в этом каталоге.

Каталог bootstrap содержит файл app.php, который загружает фреймворк. В этом каталоге также находится каталог кеша, который содержит файлы, сгенерированные платформой для оптимизации производительности, такие как файлы кеша маршрутов и служб.

Каталог config, как следует из названия, содержит все файлы конфигурации вашего приложения. Это отличная идея прочитать все эти файлы и ознакомиться со всеми доступными вам вариантами.

Каталог database содержит миграции ваших баз данных, фабрики моделей и начальные числа. При желании вы также можете использовать этот каталог для хранения базы данных SQLite.

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

Каталог resources содержит ваши представления, а также ваши необработанные, нескомпилированные ресурсы, такие как LESS, SASS или JavaScript. В этом каталоге также находятся все ваши языковые файлы.

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

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

Файл api.php содержит маршруты, которые RouteServiceProvider помещает в группу промежуточного программного обеспечения api, которая обеспечивает ограничение скорости. Эти маршруты предназначены для без 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 файле console.php вы можете определить все свои консольные команды на основе Closure. 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 точки входа (маршруты) в ваше приложение на основе консоли.

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

Каталог storage содержит ваши скомпилированные шаблоны Blade, сеансы на основе файлов, кеши файлов и другие файлы, созданные фреймворком. Этот каталог разделен на каталоги app, framework и logs. Каталог приложения может использоваться для хранения любых файлов, созданных вашим приложением. Каталог framework используется для хранения файлов и кешей, созданных платформой. Наконец, каталог logs содержит файлы logs вашего приложения.

Каталог storage / app / public может использоваться для хранения файлов, созданных пользователями, таких как аватары профиля, которые должны быть общедоступными. Вы должны создать символическую ссылку в public / storage, которая указывает на этот каталог. Вы можете создать ссылку, используя команду php artisan storage: link.

Каталог tests содержит ваши автоматизированные тесты. Пример теста PHPUnit предоставляется из коробки. Каждый тестовый класс должен иметь суффикс Test. Вы можете запускать свои тесты с помощью команд phpunit или php vendor / bin / phpunit.

Каталог Broadcasting содержит все классы каналов вещания для вашего приложения. Эти классы создаются с помощью команды make: channel. Этот каталог не существует по умолчанию, но будет создан для вас, когда вы создадите свой первый канал. Чтобы узнать больше о каналах, ознакомьтесь с документацией по трансляции событий.

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

Этот каталог Events не существует по умолчанию, но будет создан для вас Artisan-командами event: generate и make: event. Каталог событий содержит классы событий. События могут использоваться для предупреждения других частей вашего приложения о том, что произошло определенное действие, обеспечивая большую гибкость и развязку.

Каталог Exceptions содержит обработчик исключений вашего приложения, а также является хорошим местом для размещения любых исключений, создаваемых вашим приложением. Если вы хотите настроить, как ваши исключения регистрируются или обрабатываются, вам следует изменить класс Handler в этом каталоге.

Каталог Http содержит ваши контроллеры, запросы промежуточного слоя и формы. Практически вся логика обработки запросов, поступающих в ваше приложение, будет размещена в этом каталоге.

Этот каталог Jobs не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команду make: job. В каталоге заданий хранятся задания в очереди для вашего приложения. 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 команд.

Этот каталог Listeners не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команды event: generate или make: listener. Каталог Listeners содержит классы, которые обрабатывают ваши события. Слушатели событий получают экземпляр события и выполняют логику в ответ на запускаемое событие. Например, событие UserRegistered может обрабатываться приемником SendWelcomeEmail.

Этот каталог Mail не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команду make: mail. Каталог Mail содержит все ваши классы, которые представляют электронные письма, отправленные вашим приложением. Почтовые объекты позволяют вам инкапсулировать всю логику создания электронной почты в один простой класс, который может быть отправлен с помощью метода Mail :: send.

Каталог Models содержит все классы вашей модели Eloquent. Eloquent ORM, включенный в 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 новые записи в таблицу.

Этот каталог Notifications не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команду make: notification. Каталог уведомлений содержит все «транзакционные» уведомления, которые отправляет ваше приложение, например простые уведомления о событиях, которые происходят в вашем приложении. Уведомления Laravel позволяют абстрагироваться от отправки уведомлений по различным драйверам, таким как электронная почта, Slack, SMS, или хранятся в базе данных.

Этот каталог Policies не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команду make: policy. Каталог политик содержит классы политик авторизации для вашего приложения. Политики используются для определения того, может ли пользователь выполнить определенное действие с ресурсом. Для получения дополнительной информации ознакомьтесь с авторизационной документацией.

Каталог Providers содержит всех поставщиков услуг для вашего приложения. Поставщики услуг загружают ваше приложение, привязывая службы к контейнеру служб, регистрируя события или выполняя любые другие задачи для подготовки приложения к входящим запросам.

Этот каталог Rules не существует по умолчанию, но будет создан для вас, если вы выполните Artisan-команду make: rule. Каталог Rules содержит настраиваемые объекты правил проверки для вашего приложения. Правила используются для инкапсуляции сложной логики проверки в простой объект. Для получения дополнительной информации ознакомьтесь с документацией по валидации.

**Жизненный цикл запроса**

Service Providers

Одним из наиболее важных действий начальной загрузки ядра является загрузка поставщиков услуг для вашего приложения. Все поставщики услуг для приложения настраиваются в массиве поставщиков файла конфигурации config / app.php. Сначала будет вызван метод регистрации для всех провайдеров, затем, когда все провайдеры будут зарегистрированы, будет вызван метод загрузки.

Поставщики услуг несут ответственность за загрузку всех различных компонентов инфраструктуры, таких как компоненты базы данных, очереди, проверки и маршрутизации. Поскольку они загружают и настраивают каждую функцию, предлагаемую фреймворком, поставщики услуг являются наиболее важным аспектом всего процесса начальной загрузки Laravel.

#### Dispatch Request

После загрузки приложения и регистрации всех поставщиков услуг запрос будет передан маршрутизатору для отправки. Маршрутизатор отправит запрос на маршрут или контроллер, а также запустит любое связующее ПО для конкретного маршрута.

**Сосредоточьтесь на поставщиках услуг**

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

Очень важно иметь четкое представление о том, как создается и загружается приложение Laravel через поставщиков услуг. Поставщики услуг по умолчанию для вашего приложения хранятся в каталоге app / Providers.

По умолчанию AppServiceProvider довольно пуст. Этот провайдер - отличное место для добавления собственных привязок начальной загрузки вашего приложения и контейнеров служб. Для больших приложений вы можете создать несколько поставщиков услуг, каждый с более детальным типом начальной загрузки.

**Сервисный контейнер**

Сервисный контейнер Laravel - это мощный инструмент для управления зависимостями классов и выполнения внедрения зависимостей. Внедрение зависимостей - это причудливая фраза, которая по существу означает следующее: зависимости классов «вводятся» в класс через конструктор или, в некоторых случаях, методы «установщика».

Давайте посмотрим на простой пример:

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Repositories\UserRepository;

use App\Models\User;

class UserController extends Controller

{

/\*\*

\* The user repository implementation.

\*

\* @var UserRepository

\*/

protected $users;

/\*\*

\* Create a new controller instance.

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return Response

\*/

public function show($id)

{

$user = $this->users->find($id);

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

}

}

В этом примере UserController необходимо получить пользователей из источника данных. Итак, мы внедрим службу, которая может получать пользователей. В этом контексте наш UserRepository, скорее всего, использует Eloquent для извлечения информации о пользователях из базы данных. Однако, поскольку репозиторий внедрен, мы можем легко заменить его другой реализацией. Мы также можем легко «имитировать» или создать фиктивную реализацию UserRepository при тестировании нашего приложения.

**Основы привязки**

Практически все привязки вашего сервисного контейнера будут зарегистрированы у сервис-провайдеров, поэтому в большинстве этих примеров будет продемонстрировано использование контейнера в этом контексте.

**Простые привязки**

Внутри поставщика услуг у вас всегда есть доступ к контейнеру через свойство $ this-> app. Мы можем зарегистрировать привязку с помощью метода bind, передав имя класса или интерфейса, который мы хотим зарегистрировать, вместе с Closure, возвращающим экземпляр класса:

$this->app->bind('HelpSpot\API', function ($app) {

return new \HelpSpot\API($app->make('HttpClient'));

});

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

**Привязка Singleton**

Одноэлементный метод связывает класс или интерфейс с контейнером, который должен разрешаться только один раз. Как только привязка singleton разрешена, тот же экземпляр объекта будет возвращен при последующих вызовах контейнера:

$this->app->singleton('HelpSpot\API', function ($app) {

return new \HelpSpot\API($app->make('HttpClient'));

});

**Привязка экземпляров**

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

$api = new \HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\API', $api);

**Контекстная привязка**

Иногда у вас может быть два класса, которые используют один и тот же интерфейс, но вы хотите внедрить разные реализации в каждый класс. Например, два контроллера могут зависеть от разных реализаций контракта Illuminate \ Contracts \ Filesystem \ Filesystem. Laravel предоставляет простой и понятный интерфейс для определения этого 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');

});

**Связывание примитивов**

Иногда у вас может быть класс, который получает некоторые внедренные классы, но также требует внедренного примитивного значения, такого как целое число. Вы можете легко использовать контекстную привязку, чтобы ввести любое значение, которое может понадобиться вашему классу:

$this->app->when('App\Http\Controllers\UserController')

->needs('$variableName')

->give($value);

Иногда класс может зависеть от массива помеченных экземпляров. Используя метод giveTagged, вы можете легко внедрить все привязки контейнера с этим тегом:

$this->app->when(ReportAggregator::class)

->needs('$reports')

->giveTagged('reports');

**Написание поставщиков услуг**

Все поставщики услуг расширяют класс Illuminate\Support\ServiceProvider. Большинство поставщиков услуг содержат register и boot метод. В рамках метода register вы должны только привязывать вещи к сервис-контейнеру. Вы никогда не должны пытаться регистрировать какие-либо прослушиватели событий, маршруты или любую другую часть функциональности в методе register.

Artisan CLI может генерировать новый поставщик через:

php artisan make:provider RiakServiceProvider

**Метод register**

Как упоминалось ранее, в рамках метода регистрации вы должны привязывать вещи только к сервисному контейнеру. Вы никогда не должны пытаться регистрировать какие-либо прослушиватели событий, маршруты или любую другую часть функциональности в методе register. В противном случае вы можете случайно воспользоваться услугой, предоставляемой поставщиком услуг, которая еще не загружена.

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

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use Riak\Connection;

class RiakServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(Connection::class, function ($app) {

return new Connection(config('riak'));

});

}

}

Этот поставщик услуг определяет только метод register и использует этот метод для определения реализации Riak\Connection в сервис-контейнере. Если вы не понимаете, как работает сервисный контейнер, ознакомьтесь с его документацией.

#### Свойства bindings и singletons

Если ваш поставщик услуг регистрирует много 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

{

/\*\*

\* Все привязки контейнеров, которые необходимо зарегистрировать.

\*

\* @var array

\*/

public $bindings = [

ServerProvider::class => DigitalOceanServerProvider::class,

];

/\*\*

\* Все одиночные контейнеры, которые необходимо зарегистрировать.

\*

\* @var array

\*/

public $singletons = [

DowntimeNotifier::class => PingdomDowntimeNotifier::class,

ServerProvider::class => ServerToolsProvider::class,

];

}

**Метод Boot**

Итак, что, если нам нужно зарегистрировать [view composer](https://laravel.com/docs/8.x/views" \l "view-composers) в нашем сервис-провайдере? Это должно быть сделано в рамках метода boot. Этот метод вызывается после того, как все другие поставщики услуг были зарегистрированы, то есть у вас есть доступ ко всем другим услугам, которые были зарегистрированы фреймворком:

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider

{

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

view()->composer('view', function () {

//

});

}

}

## [Registering Providers](https://laravel.com/docs/8.x/providers" \l "registering-providers)

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

Чтобы зарегистрировать своего провайдера, добавьте его в массив:

'providers' => [

// Other Service Providers

App\Providers\ComposerServiceProvider::class,

],

**Отложенные Провайдеры**

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

<?php

namespace App\Providers;

use Illuminate\Contracts\Support\DeferrableProvider;

use Illuminate\Support\ServiceProvider;

use Riak\Connection;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider

{

/\*\*

\* Register any application services.

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(Connection::class, function ($app) {

return new Connection($app['config']['riak']);

});

}

/\*\*

\* Get the services provided by the provider.

\*

\* @return array

\*/

public function provides()

{

return [Connection::class];

}

}

# **Facades**

Фасады предоставляют «статический» интерфейс для классов, доступных в сервисном контейнере приложения. Laravel поставляется с множеством фасадов, которые обеспечивают доступ почти ко всем функциям Laravel. Фасады Laravel служат «статическими прокси» для базовых классов в сервисном контейнере, обеспечивая преимущества краткого выразительного синтаксиса при сохранении большей тестируемости и гибкости, чем традиционные статические методы.  
  
Все фасады Laravel определены в пространстве имен Illuminate\Support\Facades. Итак, мы можем легко получить доступ к такому фасаду:

use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {

return Cache::get('key');

});

**Когда использовать фасады?**

У фасадов много 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 протестировать.  
  
Однако при использовании фасадов необходимо соблюдать некоторую осторожность. Основная опасность фасадов - это расползание класса. Поскольку фасады очень просты в использовании и не требуют инъекций, можно легко позволить вашим классам продолжать расти и использовать множество фасадов в одном классе. При использовании внедрения зависимостей этот потенциал снижается за счет визуальной обратной связи, которую дает большой конструктор о том, что ваш класс становится слишком большим. Итак, при использовании фасадов обратите особое внимание на размер вашего класса, **чтобы объем его ответственности оставался узким**.

**Как работают фасады**

В приложении Laravel фасад - это класс, который предоставляет доступ к объекту из контейнера. Техника, которая выполняет эту работу, относится к классу фасадов. Фасады Laravel и любые пользовательские фасады, которые вы создаете, будут расширять базовый класс Illuminate \ Support \ Facades \ Facade.

Базовый класс Facade использует магический метод \_\_callStatic () для откладывания вызовов от вашего фасада к объекту, разрешенному из контейнера. В приведенном ниже примере выполняется вызов кэш-системы Laravel. Взглянув на этот код, можно предположить, что статический метод get вызывается в классе Cache:

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller

{

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return Response

\*/

public function showProfile($id)

{

$user = Cache::get('user:'.$id);

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

}

}

[https://laravel.com/docs/8.x/facades#facade-class-reference](https://laravel.com/docs/8.x/facades" \l "facade-class-reference)

# **Contracts**

Контракты Laravel - это набор интерфейсов, которые определяют основные сервисы, предоставляемые фреймворком. Например, контракт Illuminate \ Contracts \ Queue \ Queue определяет методы, необходимые для постановки заданий в очередь, а контракт Illuminate \ Contracts \ Mail \ Mailer определяет методы, необходимые для отправки электронной почты.

Каждый контракт имеет соответствующую реализацию, предусмотренную структурой. Например, Laravel предоставляет реализацию очереди с множеством драйверов и реализацию почтовой программы, работающую на SwiftMailer.

Все контракты Laravel находятся в собственном репозитории GitHub. Это обеспечивает быструю справочную информацию по всем доступным контрактам, а также к единому независимому пакету, который может использоваться разработчиками пакетов.