

# **9. MVC-фреймворки для PHP. Laravel**

# Типы фреймворков

- **Микрофреймворки** — это "конструкторы" для минимальной функциональности. Фокусируются на базовых задачах веб-разработки (маршрутизация, обработка HTTP-запросов, поддержка middleware), не навязывая архитектуру приложения.
- **Крупные фреймворки** — "полноценные станки" с жесткой структурой и готовыми решениями и инструментами для решения дополнительных задач веб-разработки.

# Микрофреймворки vs Крупные фреймворки

- Основная философия
- Функциональные возможности
- Структура проекта
- Производительность
- Экосистема и сообщество
- Случаи использования

# Основная философия

- Микрофреймворки (Slim, Lumen, Flask):
  - **Минимализм:** Решают базовые задачи веб-разработки (маршрутизация, HTTP-запросы, middleware).
  - **Гибкость:** Нет строгих правил архитектуры. Разработчик сам выбирает компоненты (шаблонизаторы, ORM).
  - **Пример кода (Slim):**

php

Копировать

```
1 $app->get('/hello', function ($request, $response) {  
2     return $response->write("Привет!");  
3 });
```

# Основная философия

- Крупные фреймворки (Laravel, Symfony, Django):
  - «Всё включено»: Предоставляют готовые решения для большинства задач (MVC, ORM, миграции, аутентификация).
  - Стандартизация: Диктуют структуру проекта (например, MVC), что упрощает масштабирование.
  - Пример кода (Laravel):

php

Копировать

```
1 Route::get('/hello', function () {  
2     return view('hello'); // Используется встроенный шаблон  
3 });
```

# Функциональные возможности

Компонент	Микрофреймворки	Крупные фреймворки
Маршрутизация	Базовая поддержка	Расширенная (группы, ресурсы)
Обработка запросов	Ручная работа с <code>\$request</code> и <code>\$response</code>	Автоматическое связывание параметров, валидация
Middleware	Простая интеграция	Встроенные middleware (аутентификация, CSRF)
Базы данных	Нужно подключать сторонние ORM	Встроенные ORM (Eloquent в Laravel)
Шаблонизаторы	Нет или минимальная поддержка	Мощные шаблонизаторы (Twig, Blade)
Аутентификация	Реализуется вручную	Готовые системы (регистрация, сброс пароля)
Миграции баз данных	Отсутствуют	Встроенные миграции и сиды

# Производительность

- **Микрофреймворки:**

Быстрее из-за минимального overhead. Идеальны для высоконагруженных API и микросервисов.

*Пример:* Slim обрабатывает тысячи запросов в секунду.

- **Крупные фреймворки:**

Медленнее «из коробки», но предлагают инструменты для оптимизации (кеширование, очереди задач).

*Пример:* Laravel с кешированием шаблонов и использованием Redis.

# Структура проекта

- **Микрофреймворки:**

Нет строгой структуры. Код может быть сосредоточен в одном файле или разбит по папкам на усмотрение разработчика.

- **Крупные фреймворки:**

Строгая структура:

Копировать

```
1 app/  
2   |— Controllers/  
3   |— Models/  
4   |— Views/  
5   └─ ...  
6 config/  
7 database/  
8 routes/
```

# Экосистема и сообщество

- **Микрофреймворки:**
  - Меньше готовых решений.
  - Требуется ручная интеграция сторонних библиотек.
- **Крупные фреймворки:**
  - Богатая экосистема (пакеты, плагины).
  - Активное сообщество и подробная документация.

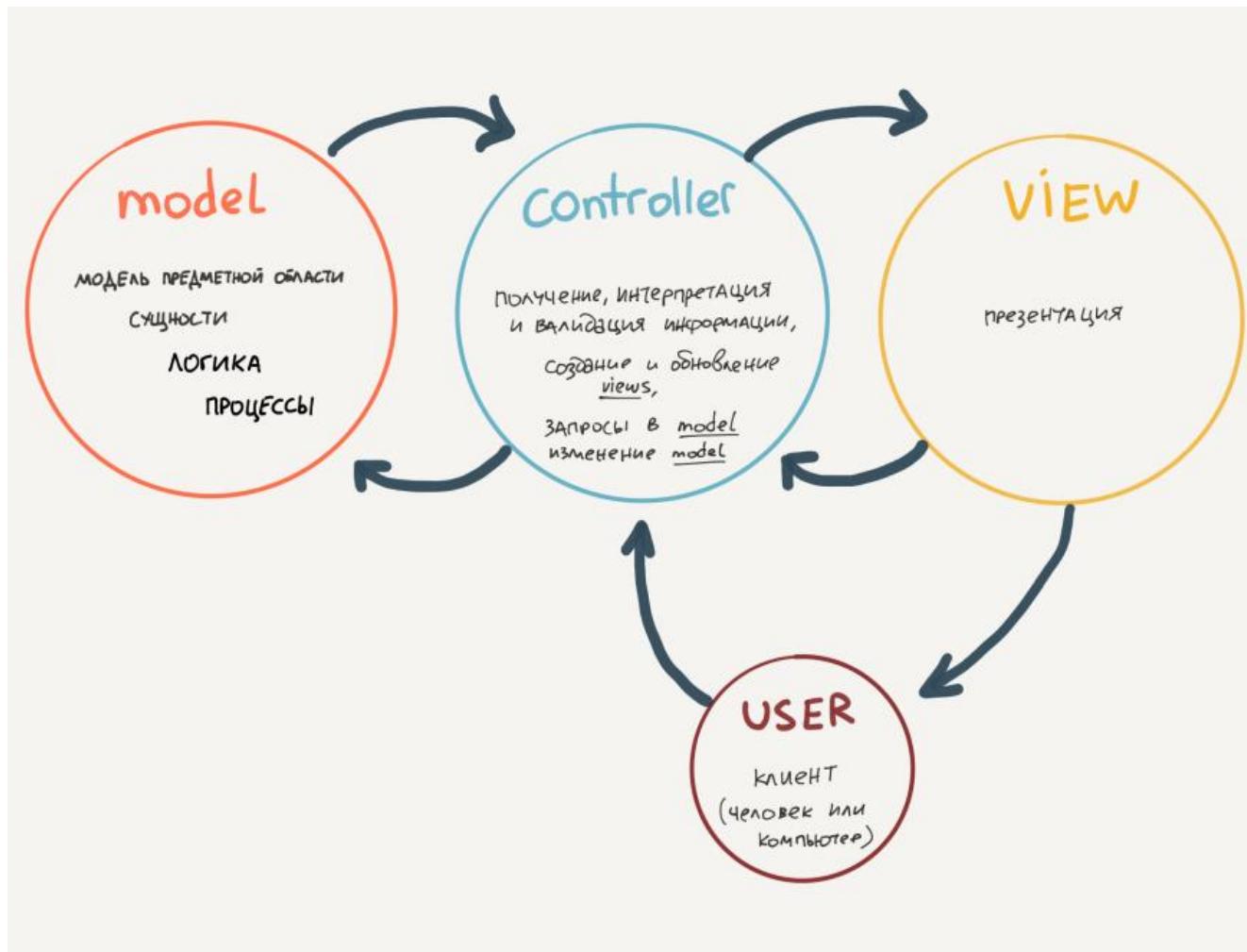
# Применимость

- **Микрофреймворки:**
  - Маленькие проекты (лендинги, API).
  - Микросервисы.
  - Прототипирование.
- **Крупные фреймворки:**
  - Корпоративные приложения (интернет-магазины, CRM).
  - Проекты с быстро растущей функциональностью.
  - Приложения с высокими требованиями к безопасности.

# Модель MVC

# Модель MVC

MVC (Model-View-Controller) — это архитектурный паттерн, используемый в веб-разработке для разделения логики приложения на три взаимосвязанные компоненты:



# Модель MVC

## 1. Model (Модель)

Отвечает за работу с данными: взаимодействие с базой данных, выполнение бизнес-логики, обработка данных. Модель не зависит от интерфейса и способов отображения.

## 2. View (Представление)

Определяет, как данные отображаются пользователю (HTML, шаблоны). Представление получает данные от контроллера и отвечает только за визуализацию.

## 3. Controller (Контроллер)

Обрабатывает входящие HTTP-запросы, связывает модель и представление. Контроллер получает данные из модели, передает их в представление и управляет потоком выполнения.

# Решение проблем смешанного подхода

## 1. Разделение компонентов

- **Model** инкапсулирует работу с данными и бизнес-логику, что упрощает её повторное использование.
- **View** отвечает только за отображение, что позволяет менять интерфейс, не затрагивая логику.
- **Controller** управляет потоком данных, изолируя обработку запросов от остального кода.

## 2. Упрощение тестирования

Компоненты тестируются изолированно. Например, можно проверить модель с помощью юнит-тестов, не запуская веб-сервер.

# Решение проблем смешанного подхода

## 3. Единая точка входа

В MVC-фреймворках (например, Laravel) все запросы проходят через один файл ( `index.php` ), что позволяет централизованно обрабатывать ошибки, аутентификацию и маршрутизацию.

## 4. Повышение безопасности

Логика обработки данных (например, экранирование SQL-запросов) сосредоточена в моделях, что снижает риск уязвимостей.

## 5. Масштабируемость

Добавление новых функций не нарушает существующий код. Например, можно создать новый контроллер для API, не трогая старые представления.

# Решение проблем смешанного подхода

Смешанный подход (PHP):

php

```
1 <?php
2 // Обработка запроса, логика и вывод в одном файле
3 $username = $_GET['user'];
4 $conn = mysqli_connect("localhost", "root", "", "db");
5 $result = mysqli_query($conn, "SELECT * FROM users WHERE name='$username'");
6 echo "<h1>Привет, " . mysqli_fetch_assoc($result)['name'] . "!</h1>";
7 ?>
```

# Решение проблем смешанного подхода: MVC

- Controller (UserController.php):

```
php Копир  
1 class UserController {  
2     public function showProfile($username) {  
3         $user = UserModel::findByName($username); // Работа с моделью  
4         return View::render('profile', ['user' => $user]); // Передача данных  
5     }  
6 }
```

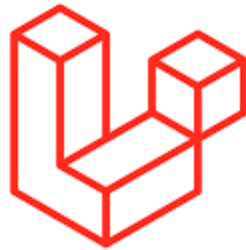
- Model (UserModel.php):

```
php Копир  
1 class UserModel {  
2     public static function findByName($name) {  
3         // Работа с БД  
4         return DB::query("SELECT * FROM users WHERE name=?", [$name]);  
5     }  
6 }
```

- View (profile.php):

```
php Копир  
1 <h1>Привет, <?= $user['name'] ?>!</h1>
```

# Фреймворк Laravel



# Laravel

Создан Тейлором Отвеллом (Taylor Otwell) в 2011 году.

Простота, мощь и современные подходы к веб-разработке

Сравнение с другими фреймворками:

- **Symfony**. Более низкоуровневый и гибкий, но требует больше ручной настройки. Laravel проще для старта.
- **CodeIgniter**. Минималистичен, но уступает в функционале (нет миграций, слабая ORM).
- **Yii2**. Мощный, но менее интуитивный синтаксис и меньшее сообщество.

# Синтаксис и структура

- **MVC-архитектура** : Четкое разделение логики (Model), представления (View) и контроллеров (Controller), что упрощает поддержку кода.
- **Читаемый код** : Laravel использует выразительный синтаксис, близкий к естественному языку (например, методы вроде `where` , `orderBy` ).

php

Копировать

```
1 // Запрос к БД через Eloquent:  
2 $users = User::where('active', 1)->orderBy('name')->get();
```

# Простота и скорость разработки

- Готовые решения "из коробки" :
  - Встроенная аутентификация (включая социальные сети), маршрутизация, миграции, кэширование, очереди.
  - Разработчики тратят меньше времени на настройку базовых функций и больше — на логику приложения.
- **Artisan CLI** : Утилита для генерации кода (моделей, контроллеров, миграций) и выполнения задач (например, очистка кэша).
- **Blade-шаблонизатор** : Простой синтаксис с наследованием шаблонов и компонентами, что ускоряет создание интерфейсов.

# Безопасность

- Встроенная защита от:
  - SQL-инъекций (благодаря использованию подготовленных запросов в Eloquent).
  - XSS-атак (автоматическая фильтрация вывода).
  - CSRF-атак (токены в формах).
  - Уязвимостей в сессиях и куках.
- Поддержка шифрования данных и безопасного хранения паролей (Bcrypt).

# Поддержка современных технологий

- **RESTful API** : Простое создание API с помощью ресурсных контроллеров и встроенного роутинга.
- **Очереди и задания** : Обработка тяжелых задач в фоне (например, отправка email) через очереди (Redis, SQS, RabbitMQ).
- **События и слушатели** : Реализация паттерна "Наблюдатель" для реакции на события в приложении.
- **Реализация SOLID и DRY** : Фреймворк поощряет соблюдение принципов чистого кода.

# Гибкость и расширяемость

- **Composer-зависимости** : Легкая интеграция сторонних библиотек.
- **Сервис-провайдеры** : Возможность подключения внешних сервисов (например, платежных систем, SMS-шлюзов).
- **Поддержка микросервисов** : Интеграция с Lagavel Octane (ускорение через Swoole/RoadRunner) и RabbitMQ.
- **Пакетная экосистема** : Огромное количество пакетов для расширения функционала (например, Lagavel Nova для админок, Telescope для отладки).

# Производительность

- **Оптимизация** : Кэширование конфигов, маршрутов и представлений.
- **Интеграция с Redis/Memcached** : Для высоконагруженных проектов.
- **Laravel Octane** : Запуск приложения в долгоживущих процессах (Swoole/RoadRunner) для максимальной скорости.

# Мощная экосистема

- **Eloquent ORM** : Удобная работа с базами данных через объекты, поддержка отношений (`hasOne`, `belongsToMany` и т.д.).
- **Laravel Mix** : Интеграция с инструментами фронтенд-разработки (Webpack, SASS, Vue.js).
- **Пакеты и сервисы** :
  - **Laravel Nova** (админ-панель),
  - **Telescope** (отладка),
  - **Horizon** (управление очередями),
  - **Vapor** (бессерверная архитектура на AWS Lambda).

# Документация и сообщество

- Детальная документация : Примеры и руководства на [официальном сайте](#) облегчают изучение.
- Сообщество :
  - Форумы (Laravel.io, Reddit),
  - Чаты (Discord, Slack),
  - Конференции (Laracon),
  - Платформа [Laracasts](#) с видеоуроками (аналог Netflix для разработчиков).

# Создание проекта в Laravel

# Процесс разработки простого приложения

- Создание каркаса приложения и начальная настройка
- Создание миграций
- Создание моделей
- Описание маршрутов и создание контроллеров для их обработки
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

- **Создание каркаса приложения и начальная настройка**
- Создание миграций
- Создание моделей
- Создание маршрутов
- Создание контроллеров
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Установка Laravel, создание приложения

- **Через Composer**

```
composer create-project laravel/laravel my_app
```

Зависимости ставятся из Packagist

- **Через установщик Laravel**

```
composer global require laravel/installer  
laravel new my_app
```

Зависимости ставятся из кэша, можно поставить дополнительные пакеты для аутентификации, фронтенда, тестирования, инициализировать Git-репозиторий.

# Установка Laravel, создание приложения

```
example-app/
├── app/ # Основные классы приложения (модели, контроллеры, сервис Копии
│   ├── Console/ # Команды Artisan (консольные команды)
│   ├── Exceptions/ # Обработка исключений
│   ├── Http/ # Контроллеры, middleware, запросы
│   │   ├── Controllers/
│   │   └── Middleware/
│   ├── Models/ # Модели Eloquent (по умолчанию их нет, создаются вручную)
│   └── Providers/ # Сервис-провайдеры (инициализация сервисов)
├── bootstrap/ # Файлы инициализации приложения
│   └── cache/ # Кэш загрузки классов
├── config/ # Конфигурационные файлы (база данных, очереди, кэш и т.д.)
├── database/ # Миграции, сидеры и фабрики
│   ├── factories/ # Фабрики моделей (для генерации тестовых данных)
│   ├── migrations/ # Миграции (структура БД)
│   └── seeders/ # Сидеры (наполнение БД тестовыми данными)
├── public/ # Публичные файлы (index.php, CSS, JS, изображения)
│   ├── css/
│   ├── js/
│   └── .htaccess # Настройки Apache (если используется)
├── resources/ # Шаблоны, языковые файлы, исходники фронтенда
│   ├── lang/ # Локализация (переводы)
│   ├── views/ # Blade-шаблоны
│   └── assets/ # SCSS, JS, изображения (компилируются в public/)
├── routes/ # Маршруты (web.php, api.php, channels.php)
├── storage/ # Логи, кэш, сессии, загруженные файлы
│   ├── app/ # Пользовательские файлы (например, загрузки)
│   ├── framework/ # Системные файлы Laravel
│   └── logs/ # Логи приложения
├── tests/ # Тесты (PHPUnit, Pest)
│   ├── Feature/ # Функциональные тесты
│   └── Unit/ # Юнит-тесты
├── vendor/ # Зависимости Composer (библиотеки)
├── .env # Переменные окружения (настройки БД, ключи)
├── artisan # Скрипт для запуска Artisan (CLI Laravel)
├── composer.json # Зависимости проекта и метаданные
└── package.json # Зависимости Node.js (для фронтенда)
```

# Установка Laravel, создание приложения

Файловая структура Laravel логична и масштабируема. Основные компоненты:

- Бэкенд : `app/` , `routes/` , `database/` .
- Фронтенд : `resources/views/` , `public/` .
- Конфигурация : `config/` , `.env` .
- Тестирование : `tests/` .

# Настройка окружения

- Файл `.env` :

Настройте параметры окружения (база данных, почта, API-ключи и т.д.):

```
env
1 APP_NAME=MyApp
2 APP_ENV=local
3 APP_DEBUG=true
4 DB_CONNECTION=mysql
5 DB_HOST=127.0.0.1
6 DB_PORT=3306
7 DB_DATABASE=myapp
8 DB_USERNAME=root
9 DB_PASSWORD=
```

- Конфигурационные файлы :

Основные настройки хранятся в `config/` (например, `config/database.php` для БД).

# Поддерживаемы базы данных "из коробки"

- MySQL (включая MariaDB)
- PostgreSQL
- SQLite
- Microsoft SQL Server

Для работы с этими СУБД достаточно настроить параметры подключения в файле `.env` и использовать стандартные драйверы.

- Создание каркаса приложения и начальная настройка
- **Создание миграций**
- Создание моделей
- Создание маршрутов
- Создание контроллеров
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Миграции в Laravel

**Миграции** — инструмент для управления структурой БД через ORM-код (замена SQL-скриптов).

- **Версионный контроль БД.** Все изменения структуры хранятся в файлах, которые можно отслеживать через Git.
- **Повторяемость.** Миграцию можно применить одной командой.
- **Откат изменений.** Если что-то пошло не так, можно откатить миграцию.
- **Автоматизация.** Нет необходимости вручную писать SQL-запросы для создания таблиц.

# Генерация миграций

Миграции создаются через Artisan (CLI Laravel).

## 1. Генерация миграции

bash

Копировать

```
1 php artisan make:migration create_posts_table --create=posts
```

- `create_posts_table` — название миграции (должно быть уникальным и описательным).
- `--create=posts` — указывает, что миграция создает таблицу `posts`.

Файл миграции появится в `database/migrations/` с именем вида

`2023_10_01_000000_create_posts_table.php`.

---

# Структура миграции

```
class CreatePostsTable extends Migration
{
    // Применение миграции (создание таблицы)
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id(); // Первичный ключ (INT AUTO_INCREMENT)
            $table->string('title'); // VARCHAR(255)
            $table->text('content')->nullable(); // TEXT (может быть NULL)
            $table->timestamps(); // created_at и updated_at
        });
    }

    // Откат миграции (удаление таблицы)
    public function down()
    {
        Schema::dropIfExists('posts');
    }
}
```

Копировать

# Выполнение миграции

Чтобы применить миграции к базе данных:

```
bash
```

Копировать

```
1 php artisan migrate
```

Laravel создаст таблицы и запишет информацию о выполненных миграциях в таблицу `migrations` .

# Откат миграций

- Откат последней миграции :

```
bash
```

```
1 php artisan migrate:rollback
```

- Полный откат всех миграций :

```
bash
```

```
1 php artisan migrate:reset
```

- Пересоздание всех таблиц (удаление и повторный запуск миграций):

```
bash
```

```
1 php artisan migrate:fresh
```

# Служебные таблицы

При выполнении команды:

```
bash
```

Копирс

```
1 php artisan migrate
```

Laravel создает **служебные таблицы**, которые нужны для работы фреймворка:

- **migrations**: Хранит историю выполненных миграций (чтобы избежать дублирования).
- **users**: Таблица для пользователей (логин, пароль, email и т.д.).
- **password\_reset\_tokens**: Для хранения токенов сброса пароля.
- **sessions**: Хранение сессий (если используется драйвер БД для сессий).
- **cache**, **jobs**, **failed\_jobs**: Таблицы для кэширования, очередей и отслеживания ошибок в очередях.

- Создание каркаса приложения и начальная настройка
- Создание миграций
- **Создание моделей**
- Создание маршрутов
- Создание контроллеров
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Модели в Laravel

**Модели** в Laravel — это классы, которые представляют данные из таблиц базы данных и взаимодействуют с ними через Eloquent ORM (объектно-реляционное отображение).

Eloquent реализует паттерн проектирования Active Record, при котором объект напрямую связан с таблицей в БД:

- **Самоидентификация.** Объект "знает" свою таблицу и структуру.
- **CRUD-методы** встроены в объект.
- **Атрибуты.** Свойства объекта соответствуют полям таблицы.

Альтернативный подход — паттерн Data Mapper, когда объекты данных отделены от логики доступа к БД. Для работы с данными используется отдельный класс-маппер.

# Связь модели с таблицей

По умолчанию модель связывается с таблицей, название которой является множественной формой от названия модели в нижнем регистре (например, модель `Post` → таблица `posts` ).

## Явное указание таблицы :

Если таблица имеет другое название, укажите его в свойстве `$table` :

php

Копировать

```
1 namespace App\Models;
2
3 use Illuminate\Database\Eloquent\Model;
4
5 class Post extends Model
6 {
7     protected $table = 'blog_posts'; // Связь с таблицей blog
8 }
```

# Пространство имен Illuminate

---

`Illuminate` — это название пространства имен (namespace), которое используется в `Laravel` для большинства его внутренних компонентов и ядра фреймворка. По сути, это "сердце" `Laravel`, где находятся все основные классы и функции, управляющие работой приложения.

## Зачем это нужно?

- **Организация кода:** Все компоненты `Laravel` группируются под пространством имен `Illuminate`, чтобы избежать конфликтов имен.
  - **Автономность:** Многие компоненты (например, `illuminate/support`, `illuminate/database`) могут использоваться отдельно от `Laravel` в других проектах.
  - **Расширяемость:** Разработчики могут переопределять или расширять классы из `Illuminate`, создавая свои сервис-провайдеры или фасады.
-

# Создание модели

Модели создаются через **Artisan** (CLI Laravel).

**Базовая команда :**

```
bash
```

```
1 php artisan make:model Название_модели
```

**Пример:**

```
bash
```

```
1 php artisan make:model Post
```

Файл модели будет создан в `app/Models/Post.php`.

# Создание модели

---

- Создать модель с миграцией :

```
bash
```

```
1 php artisan make:model Post -m
```

- Создать модель с миграцией, фабрикой и сидером :

```
bash
```

```
1 php artisan make:model Post -mf
```

- 
- **Фабрики** — это шаблоны для создания отдельных моделей с реалистичными данными.
  - **Сидеры** — это скрипты для массового создания данных с использованием фабрик.
  - Вместе они упрощают тестирование и разработку, избавляя от ручного ввода данных.

# Работа с данными через модель

---

Создание записи :

php

```
1 $post = new Post();
2 $post->title = 'Новый пост';
3 $post->content = 'Содержание поста';
4 $post->save();
```

Или через массовое присвоение:

php

```
1 Post::create([
2     'title' => 'Новый пост',
3     'content' => 'Содержание',
4 ]);
```

# Работа с данными через модель

---

## Обновление данных :

php

```
1 $post = Post::find(1);  
2 $post->title = 'Обновленный заголовок';  
3 $post->save();
```

## Удаление данных :

php

```
1 $post = Post::find(1);  
2 $post->delete();
```

# Работа с данными через модель

## Получение данных :

php

Копировать

```
1 // Все посты:
2 $posts = Post::all();
3
4 // Пост по ID:
5 $post = Post::find(1);
6
7 // Посты с условием:
8 $posts = Post::where('active', 1)->orderBy('created_at')->get();
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $table = 'posts';

    protected $fillable = ['title', 'content', 'author_id'];

    // Отношение "Один ко многим" (пост имеет много комментариев)
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }

    // Отношение "Многие к одному" (пост принадлежит автору)
    public function author()
    {
        return $this->belongsTo(User::class, 'author_id');
    }

    // Мутатор для поля title
    public function setTitleAttribute($value)
    {
        $this->attributes['title'] = ucfirst($value);
    }
}
```

- Создание каркаса приложения и начальная настройка
- Создание миграций
- Создание моделей
- **Создание маршрутов**
- Создание контроллеров
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Маршруты в Laravel

Маршруты (Routes) в Laravel определяют, как приложение реагирует на запросы к конкретным URL-адресам. Они связывают URI (например, `/about`) с контроллерами, замыканиями (анонимными функциями) или действиями, которые обрабатывают запрос и возвращают ответ.

---

## Где настраиваются маршруты?

Маршруты хранятся в файлах в папке `routes` :

- `web.php` — маршруты для веб-интерфейса (сессии, CSRF-защита, кэширование).
- `api.php` — маршруты для API (без защиты CSRF, обычно возвращают JSON).
- `console.php` — маршруты для Artisan-команд (редко используется).

# Маршруты в Laravel

В Laravel маршруты (routes) определяются в файлах `routes/web.php` и `routes/api.php`. Для работы с ними используется фасад `Route`. Примеры:

## 1. Базовые маршруты

php

Копировать

```
1 use Illuminate\Support\Facades\Route;
2
3 // GET-запрос к корню сайта
4 Route::get('/', function () {
5     return view('welcome'); // Возвращает представление
6 });
7
8 // GET-запрос к /about
9 Route::get('/about', function () {
10    return 'Страница "О нас"';
11 });
12
13 // POST-запрос к /submit
14 Route::post('/submit', function () {
15    return 'Форма отправлена!';
16 });
```

Фасады (Facades) в Laravel — это удобный способ доступа к функционалу фреймворка через **статические методы**, скрывающие сложность взаимодействия с сервис-контейнером. Они упрощают работу с классами, реализованными в контейнере зависимостей.

---

## Как это работает?

### 1. Статический интерфейс к динамическим объектам :

- Фасады выглядят как статические классы, но на самом деле они динамически разрешают зависимости через сервис-контейнер.
- Пример: `Cache::get('key')` обращается к экземпляру класса, управляющего кэшем.

### 2. Метод `__callStatic()` :

- Когда вы вызываете `SomeFacade::method()`, PHP перехватывает вызов через магический метод `__callStatic()`.
- Фасад находит нужный объект в контейнере и вызывает его метод.

## Примеры встроенных фасадов

ФАСАД	ОПИСАНИЕ	ПРИМЕР ИСПОЛЬЗОВАНИЯ
<code>DB</code>	Работа с БД (Query Builder)	<code>DB::table('users')-&gt;get()</code>
<code>Auth</code>	Аутентификация	<code>Auth::user()</code>
<code>Cache</code>	Кэширование данных	<code>Cache::remember('key', \$ttl, ...)</code>
<code>Event</code>	События	<code>Event::dispatch(new UserRegistered)</code>
<code>Storage</code>	Работа с файлами	<code>Storage::disk('s3')-&gt;put(...)</code>

## Преимущества фасадов

- **Удобство** : Короткий и понятный синтаксис.
  - **Читаемость** : Код выглядит декларативно.
  - **Тестируемость** : Можно заменять реализации через контейнер (например, моки в тестах).
- 

## Недостатки

- **Скрытие зависимостей** : Сложно понять, от каких классов зависит код.
- **Злоупотребление статикой** : Может нарушить принципы SOLID, если фасады используются в бизнес-логике.
- **Сложность рефакторинга** : Изменение реализации требует обновления всех вызовов фасада.

# Маршруты в Laravel

Чтобы связать маршрут с методом контроллера, укажите класс и метод:

php

Копирова

```
1 use App\Http\Controllers\PostController;
2 use Illuminate\Support\Facades\Route;
3
4 // GET-запрос к /posts вызывает метод index контроллера PostController
5 Route::get('/posts', [PostController::class, 'index']);
6
7 // POST-запрос к /posts/store вызывает метод store
8 Route::post('/posts/store', [PostController::class, 'store']);
```

- Создание каркаса приложения и начальная настройка
- Создание миграций
- Создание моделей
- Создание маршрутов
- **Создание контроллеров**
- Создание выходных представлений
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Контроллеры в Laravel

Контроллеры в Laravel — это классы, которые обрабатывают HTTP-запросы, взаимодействуют с моделями (базой данных) и возвращают ответы (например, HTML-страницы или JSON). Они играют ключевую роль в архитектуре MVC (Model-View-Controller), разделяя логику приложения на слои.

---

## Зачем нужны контроллеры?

- **Группировка логики:** Контроллеры объединяют связанные методы (например, все операции с пользователями).
- **Чистота кода:** Убирают логику из маршрутов ( `routes/web.php` ), делая код структурированным.
- **Повторное использование:** Методы контроллеров можно вызывать из разных маршрутов.

# Создание контроллеров

Используйте Artisan-команду:

```
bash
```

```
1 php artisan make:controller ИмяКонтроллера
```

Пример:

```
bash
```

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

Это создаст файл `UserController.php` в `app/Http/Controllers` :

```
php
```

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 class UserController extends Controller
8 {
9     // Здесь будут методы
10 }
```

# Контроллеры в Laravel

## 1. Простой контроллер

Содержит произвольные методы:

php

Копировать

```
1 class UserController extends Controller {
2     public function show($id) {
3         return "Просмотр пользователя с ID: $id";
4     }
5
6     public function create() {
7         return view('user.create');
8     }
9 }
```

Привязка к маршруту:

php

Копировать

```
1 // routes/web.php
2 use App\Http\Controllers\UserController;
3
4 Route::get('/user/{id}', [UserController::class, 'show']);
5 Route::get('/user/create', [UserController::class, 'create']);
```

## 2. Ресурсный контроллер (RESTful)

Создает CRUD-методы для работы с ресурсом (например, статьями):

bash

Копировать

```
1 php artisan make:controller PostController --resource
```

Генерирует методы:

php

Копировать

```
1 class PostController extends Controller {
2     public function index()    { /* Список ресурсов */ }
3     public function create()  { /* Форма создания */ }
4     public function store()   { /* Сохранение */ }
5     public function show($id) { /* Просмотр */ }
6     public function edit($id) { /* Форма редактирования */ }
7     public function update($id) { /* Обновление */ }
8     public function destroy($id) { /* Удаление */ }
9 }
```

Привязка к маршруту:

php

Копировать

```
1 Route::resource('posts', PostController::class);
```

API-контроллеры в Laravel — это специализированные контроллеры, предназначенные для обработки запросов к API. Они отличаются от обычных (веб-)контроллеров структурой, поведением и назначением.

---

## Зачем нужны API-контроллеры?

### 1. Для создания RESTful API

Предоставляют данные в формате JSON/XML, а не HTML-страницы.

### 2. Для взаимодействия с внешними сервисами

Мобильные приложения, SPA (React/Vue), сторонние интеграции.

### 3. Упрощение разработки

Генерируют стандартные методы для CRUD-операций, поддерживают версионирование API.

# Создание API-контроллера

Используйте Artisan с флагом `--api`:

```
bash
```

```
1 php artisan make:controller API/PostController --api
```

Сгенерированные методы:

```
php
```

```
1 class PostController extends Controller {
2     public function index()      { /* Все записи */ }
3     public function store()     { /* Создание */ }
4     public function show($id)   { /* Просмотр */ }
5     public function update($id) { /* Обновление */ }
6     public function destroy($id){ /* Удаление */ }
7 }
```

# Подключение API-контроллера

В файле `routes/api.php` :

php

```
1 use App\Http\Controllers\API\PostController;  
2  
3 // Ресурсный маршрут для API  
4 Route::apiResource('posts', PostController::class);
```

Генерируемые маршруты:

HTTP-МЕТОД	URI	ДЕЙСТВИЕ
GET	/api/posts	index
POST	/api/posts	store
GET	/api/posts/{id}	show
PUT/PATCH	/api/posts/{id}	update
DELETE	/api/posts/{id}	destroy

- Создание каркаса приложения и начальная настройка
- Создание миграций
- Создание моделей
- Создание маршрутов
- Создание контроллеров
- **Создание выходных представлений**
- Реализация бизнес-логики
- Запуск с помощью встроенного сервера

# Представления в Laravel

Представления (Views) в Laravel — это шаблоны, которые отвечают за отображение данных пользователю. Они разделяют логику приложения (контроллеры, модели) и HTML/CSS/JS, что упрощает поддержку кода. Laravel использует движок шаблонов **Blade** для создания представлений.

---

## Где хранятся представления?

Представления находятся в папке `resources/views`. Например:

- `resources/views/welcome.blade.php` — главная страница.
  - `resources/views/users/index.blade.php` — список пользователей.
-

# Создание представлений

## 1. Вручную:

Создайте файл с расширением `.blade.php` в папке `resources/views`.

Пример: `resources/views/hello.blade.php`:

```
html Копировать
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Приветствие</title>
5 </head>
6 <body>
7   <h1>Добро пожаловать, {{ $name }}!</h1>
8 </body>
9 </html>
```

## 2. Через Artisan (необязательно):

Стандартной команды для создания представлений нет, но можно создать свою команду или использовать:

```
bash Копировать
1 php artisan make:view hello
```

# Отображение представлений

Через метод `view()` в контроллере или маршруте:

php

Копировать

```
1 // В контроллере
2 public function showWelcome() {
3     return view('welcome'); // resources/views/welcome.blade.php
4 }
5
6 // В маршруте (routes/web.php)
7 Route::get('/hello', function () {
8     return view('hello', ['name' => 'Алекс']);
9 });
```

Контроллер:

```
php
1 namespace App\Http\Controllers;
2
3 use App\Models\Post;
4 use Illuminate\Http\Request;
5
6 class PostController extends Controller {
7     public function index() {
8         $posts = Post::all();
9         return view('posts.index', compact('posts'));
10    }
11 }
```

Представление ( `resources/views/posts/index.blade.php` ):

```
blade
1 @extends('layouts.app')
2
3 @section('content')
4     <h1>Список статей</h1>
5     @foreach ($posts as $post)
6         <div class="post">
7             <h2>{{ $post->title }}</h2>
8             <p>{{ $post->excerpt }}</p>
9         </div>
10    @endforeach
11 @endsection
```

- Создание каркаса приложения и начальная настройка
- Создание миграций
- Создание моделей
- Создание маршрутов
- Создание контроллеров
- Создание выходных представлений
- **Реализация бизнес-логики**
- Запуск с помощью встроенного сервера

# Где размещать бизнес-логику?

- Контроллеры
- Модели
- Дополнительные классы

# Бизнес-логика в контроллерах

- Что можно делать:
  - Принимать входящие запросы (GET/POST-параметры).
  - Вызывать сервисы или модели для обработки данных.
  - Возвращать ответы (JSON, представления).
- Что нельзя делать:
  - Писать сложную логику (например, расчеты, взаимодействие с несколькими моделями).
  - Работать напрямую с БД (лучше через модели или репозитории).

## Пример плохого кода (толстый контроллер):

```
php Копировать  
1 class UserController extends Controller {  
2     public function store(Request $request) {  
3         // Валидация, работа с БД, отправка email – всё в контроле  
4         $validated = $request->validate([...]);  
5         $user = User::create($validated);  
6         Mail::to($user)->send(new WelcomeEmail());  
7         return redirect()->route('users.index');  
8     }  
9 }
```

## Правильный код (тонкий контроллер):

```
php Копировать  
1 class UserController extends Controller {  
2     protected $userService;  
3  
4     public function __construct(UserService $userService) {  
5         $this->userService = $userService;  
6     }  
7  
8     public function store(Request $request) {  
9         $user = $this->userService->createUser($request->all());  
10        return redirect()->route('users.index');  
11    }  
12 }
```

# Бизнес-логика в моделях (Eloquent)

- Что можно делать:
  - Методы для работы с данными (CRUD).
  - Связи между таблицами (отношения).
  - Локальная логика, специфичная для модели (например, `User::getFullName()`).
- Что нельзя делать:
  - Логику, затрагивающую несколько моделей.
  - Валидацию (лучше использовать FormRequest).
  - Бизнес-правила, не связанные напрямую с моделью.

# Бизнес-логика в моделях (Eloquent)

```
class User extends Model {  
    // Связь с постами  
    public function posts() {  
        return $this->hasMany(Post::class);  
    }  
  
    // Локальный метод  
    public function getFullName() {  
        return $this->first_name . ' ' . $this->last_name;  
    }  
}
```

# **Бизнес-логика в сервисных классах (Service Layer)**

---

- Когда использовать:
  - Логика, затрагивающая несколько моделей.
  - Сложные операции (например, создание заказа с резервированием товаров).
  - Повторяющаяся логика (чтобы избежать дублирования).
- Преимущества:
  - Тестируемость.
  - Повторное использование кода.

# Принцип "Тонкий контроллер - толстая модель"

- Суть:

Контроллер должен быть минимальным (только обработка запроса/ответа), а бизнес-логика — в моделях или сервисах.

- Когда это работает:

- В небольших проектах.
- Для простой логики, специфичной для одной модели.

# Принцип "Тонкий контроллер - толстая модель"

---

```
// Тонкий контроллер
class PostController extends Controller {
    public function store(Request $request) {
        $post = Post::create($request->validated());
        return redirect()->route('posts.show', $post);
    }
}

// Толстая модель
class Post extends Model {
    protected static function boot() {
        parent::boot();
        static::creating(function ($post) {
            $post->slug = Str::slug($post->title);
        });
    }
}
```

# Бизнес-логика в репозитории

---

- Зачем нужны:
  - Абстрагирование доступа к данным (например, для замены БД без изменения бизнес-логики).
  - Работа с кэшем, сложными запросами.

```
interface UserRepository {
    public function find($id);
    public function create(array $data);
}

class EloquentUserRepository implements UserRepository {
    public function find($id) {
        return User::find($id);
    }

    public function create(array $data) {
        return User::create($data);
    }
}
```

# Бизнес-логика в сервисных классах (Service Layer)

---

```
namespace App\Services;

class UserService {
    public function createUser(array $data) {
        $validated = Validator::make($data, [...])->validate();
        $user = User::create($validated);
        event(new UserCreated($user)); // Отправка события
        return $user;
    }
}
```

# Размещение бизнес-логики

- Контроллеры — только для обработки запросов.
- Модели — для базовой логики, связанной с данными.
- Сервисы — для сложной, многокомпонентной логики.
- Репозитории — для абстракции доступа к данным.

Принцип тонких контроллеров справедлив, но в больших проектах бизнес-логику стоит выносить в отдельные слои (сервисы, события, репозитории). Это улучшает тестируемость, поддерживаемость и масштабируемость кода.