

LARAVEL. Правила разработки в SBI

Naming conventions.....	4
Таблицы.....	4
Общие правила.....	4
Промежуточные таблицы (pivot).....	4
Название полей в таблице.....	4
Внешние ключи.....	5
Классы.....	5
Общие правила.....	5
Модель.....	5
Переменные.....	5
Общие правила.....	5
Константы.....	6
Массивы.....	6
Название методов в классах.....	6
Общие правила.....	6
Связи в моделях.....	6
Название маршрутов.....	7
Общие правила.....	7
Правила организации файлов и папок в Laravel.....	8
Services.....	8
Controllers, Resources и Requests.....	8
Правила для описания маршрутов (Routes) и создания контроллеров (Controllers) с методами.....	9
Migrations.....	9
Models.....	9
Seeders.....	9
Strategy Pattern.....	9
Service Pattern.....	11
Общее описание.....	11
Создание Service-класса.....	11
Вызов Service-класса в контроллере.....	12
Регистрация Service-классов (необязательно).....	13

Вызов сервиса в другом сервисе.....	13
Генерация API документа.....	14
Архитектура обработки запросов LARAVEL.....	14
Последовательность обработки запроса.....	14
Маршрут (Route).....	15
Контроллер (Controller).....	15
Запрос и правила (Request-Rules).....	16
Сервис (Service).....	16
Ресурс (Resource).....	17
Формирование сущностей в Laravel: Пошаговое руководство.....	18
Общий путь.....	18
MIGRATION.....	18
Создание миграции.....	18
Определение структуры таблицы.....	18
Применение миграции.....	19
MODEL.....	19
Создание модели.....	19
FACTORY.....	19
Создание фабрики.....	19
Определение фабрики.....	20
SERVICE.....	20
Создание сервиса.....	20
Использование сервиса.....	20
SEEDER.....	21
Создание сидера.....	21
Определение сидера.....	21
Запуск сидера.....	21
Важные утилиты.....	22
Генерация моделей (DEV).....	22
Файловая библиотека SPATIE.....	22
Работа с EXCEL документами.....	22
Мультиязычность AstroTomic.....	22
Логирование SPATIE.....	22
Отладка ошибок Telescope.....	22
Локальный запуск в Docker.....	22
SOLID (изучение процесса).....	23
Single Responsibility Principle (SRP).....	23
Open/Closed Principle (OCP).....	23
Liskov Substitution Principle (LSP).....	24

Interface Segregation Principle (ISP).....	24
Dependency Inversion Principle (DIP).....	25
Автоматизация и оптимизация разработки на начальном этапе (процесс описания...)	26
Генерация сущностей в EXCEL.....	26
Генерация API методов в EXCEL.....	26

Naming conventions

Таблицы

Общие правила

Используйте строчные буквы и множественное число.

Примеры: `users`, `orders`, `user_transactions`

Промежуточные таблицы (pivot)

Имя промежуточных таблиц: Используйте единственное число для имен промежуточных таблиц, соединяющих две сущности, и сортируйте их в алфавитном порядке.

Aa [eɪ]	Bb [biː]	Cc [siː]	Dd [diː]
Ee [iː]	Ff [ef]	Gg [dʒiː]	Hh [eɪtʃ]
Ii [aɪ]	Jj [dʒeɪ]	Kk [keɪ]	Ll [el]
Mm [em]	Nn [en]	Oo [ou]	Pp [pɪ]
Qq [kjuː]	Rr [ɑː]	Ss [es]	Tt [tiː]
Uu [juː]	Vv [viː]	Ww [ˈdʌbljuː]	Xx [eks]
	Yy [waɪ]	Zz [zed]	

Пример: `role_user`, `category_product`

Название полей в таблице

Стиль названий: Поля в таблицах также должны быть в строчных буквах и разделяться символом подчеркивания “_”.

Пример: `first_name`, `last_name`, `created_at`, `updated_at`

Внешние ключи

Стиль именования: Названия внешних ключей должны следовать формату `{имя_модели}_id`, например:

`user_id` (для связи с таблицей `users`)

Классы

Общие правила

Стиль именования классов: Используйте **PascalCase** для имен классов

Пример: `UserController`, `User`, `UserJob`, `StoreUserRequest`

Модель

Имена моделей должны соответствовать названиям таблиц.

Пример:

`User` для таблицы `users`

`Order` для таблицы `orders`

`UserTransaction` для таблицы `user_transactions`

Переменные

Общие правила

Используйте **camelCase** для имен переменных в коде, начиная с маленькой буквы.

`$firstName = 'Denis';`

Константы

Для констант используйте стиль `UPPER_SNAKE_CASE`. Это поможет четко выделить их из обычных переменных.

```
const MAX_ATTEMPTS = 5;
```

Массивы

При работе с массивами используйте множественные числа и имена переменных, которые четко описывают содержимое массива.

```
$userRoles = ['admin', 'editor', 'viewer'];
```

Название методов в классах

Общие правила

Для именования методов в Laravel принято использовать стиль **camelCase** – когда название начинается с маленькой буквы, а каждое следующее слово с заглавной.

Пример:

```
getUserTransactions()
```

```
createUserTransaction()
```

Связи в моделях

Связи в моделях реализуются как методы, поэтому к ним применяются те же правила, что и к любым другим методам. Для наглядности приведем примеры:

```
class User extends Model
{
    public function userTransactions()
    {
        return $this->hasMany(UserTransaction::class);
    }
}
```

```
class User extends Model
{
    public function userTransaction()
    {
        return $this->hasOne(UserTransaction::class);
    }
}

class UserTransaction extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Название маршрутов

Общие правила

Для формирования роута, необходимо использовать **kebab-case** (с разделением слов через дефисы) для маршрутов API. Это соответствует RESTful соглашениям и является лучшей практикой в большинстве случаев.

Пример:

```
/api/v1/finance-transactions
```

Правила организации файлов и папок в Laravel

Services

Сервисы следует разделять на две папки: **Api** и **System**. Сервисы, отвечающие за управление моделями, следует размещать в главном каталоге **Services**.

Пример:

`.../Services/Api/TelegramService.php` – для внешних интеграций.

`.../Services/System/TableInfoService.php` – для системных функций и утилит.

`.../Services/SomeModelService.php` – для работы с моделями.

Controllers, Resources и Requests

Контроллеры, ресурсы и запросы необходимо хранить в одинаковых папках, соблюдая единый неймспейс. Пример структуры:

`.../Controllers/v1/Admin/UserController`

`.../Resources/v1/Admin/UserResource`

`.../Requests/v1/Admin/StoreUserRequest`

`.../Requests/v1/Admin/UpdateUserRequest`

В данном примере `v1/Admin` должен оставаться одинаковым во всех частях.

Правила для описания маршрутов (Routes) и создания контроллеров (Controllers) с методами

Роуты должны интуитивно указывать на расположение контроллера в структуре проекта. Например:

- Если маршрут расположен, `[GET] v1/admin/users`, то в контроллере должно быть:
`.../Controllers/v1/Admin/UserController@index`
- Если маршрут, например, `[POST] v1/admin/auth/login`, то в контроллере должно быть:
`.../Controllers/v1/Admin/AuthController@login`

Migrations

Миграции необходимо хранить на одном уровне без подкаталогов.

Models

Модели необходимо хранить на одном уровне без подкаталогов.

Seeders

Сидеры необходимо хранить на одном уровне без подкаталогов.

Strategy Pattern

Если в коде присутствует множество условий `if`, `else` или конструкций `switch case`, следует рассмотреть возможность использования структур данных для упрощения логики. Такой подход позволяет избежать громоздкости и сложных ветвлений, что значительно упрощает поддержку и уменьшает вероятность ошибок. Вместо того чтобы изменять основную логику кода, достаточно дополнить массив или объект новыми данными. Это делает систему более адаптивной к новым требованиям и облегчает внесение изменений, не требуя переписывания или усложнения существующего кода.

```

class SomeClass
{
    protected $countryCodes = [
        '998' => ['90', '92', '93'], // коды для Узбекистана
        '992' => ['96', '11'], // коды для Таджикистана
    ];

    public function validate(string $attribute, mixed $value, \Closure
$fail): void
    {
        // Удаляем знак + и пробелы, если они есть
        $value = preg_replace('/^\+|\s/', '', $value);
        // Извлекаем код страны (первые 3 символа)
        $countryCode = substr($value, 0, 3);
        // Проверяем, существует ли код страны
        if (!isset($this->countryCodes[$countryCode])) {
            echo "Номер телефона '{$value}' не соответствует правилам для
указанной страны.";
            return; // Прерываем выполнение, если код страны не найден
        }
        // Формируем регулярное выражение динамически
        $validCodes = implode('|', $this->countryCodes[$countryCode]);
        $regex = "/^{$countryCode}({$validCodes})\d{7}$/";
        // Проверяем соответствие номера правилам
        if (!preg_match($regex, $value)) {
            echo "Номер телефона '{$value}' не соответствует правилам для
указанной страны.";
        }
    }
}

```

Service Pattern

Общее описание

Service Pattern в Laravel – это архитектурный подход, который помогает разделить бизнес-логику на отдельные сервисы, что делает код более структурированным и поддерживаемым. Этот паттерн полезен, когда нужно вынести сложные операции из контроллеров или моделей в отдельные классы, чтобы облегчить их тестирование и повторное использование.

!!! Мы приняли решение **отказаться** от использования **Repository Pattern** из-за избыточности и создания лишней абстракции, которая увеличивает время разработки. Вместо этого, мы будем сосредотачиваться на использовании **Service Pattern**, что позволит избежать ненужной прослойки и упростить код.

Вот пошаговое описание, как реализовать **Service Pattern** в Laravel:

Создание Service-класса

Создавать Service-классы в папке `app/Services`. Эти классы будут содержать бизнес-логику.

`UserService.php` в папке `Services`:

```
<?php

namespace App\Services;

use App\Models\User;

class UserService
{
    public function createUser(array $data): User //
    UserCreateRequest to make sure that data is validated and accepts
    only required filtered fields.
```

```

    {
        // Здесь может быть сложная логика создания пользователя
        return User::create($data);
    }

    public function updateUser(User $user, array $data): User
    {
        // UserUpdateRequest for the same reason
        // Логика обновления пользователя
        $user->update($data);
        return $user;
    }

    // Другие методы, связанные с пользователями
}

```

Вызов Service-класса в контроллере

Теперь контроллер будет использовать сервис для выполнения бизнес-логики. Это делает контроллеры более чистыми и простыми.

UserController.php:

```

<?php

namespace App\Http\Controllers;

use App\Services\UserService;
use Illuminate\Http\Request;
use App\Models\User;

class UserController extends Controller
{
    protected $userService;

    // Инъекция зависимости UserService через конструктор
    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }
}

```

```

    public function store(Request $request)
    {
        // Here is also Request is not validated and there is
        danger of feeding service with wrong data
        // Логика создания пользователя теперь находится в сервисе
        $user = $this->userService->createUser($request->all());

        return response()->json($user, 201);
    }

    public function update(Request $request, User $user)
    {
        // Логика обновления пользователя через сервис
        $user = $this->userService->updateUser($user,
        $request->all());

        return response()->json($user);
    }
}

```

Регистрация Service-классов (необязательно)

Обычно Laravel автоматически внедряет сервисы через dependency injection, но для явной регистрации сервисов можно использовать **Service Provider**.

В `app/Providers/AppServiceProvider.php`:

```

public function register()
{
    // Binding to interfaces to rather than to concrete
    implementation is encouraged By Solid Rules UserServiceInterface
    $this->app->singleton(UserService::class, function ($app) {
        return new UserService();
    });
}

```

Вызов сервиса в другом сервисе

Если один сервис требует использование другого:

```
class SomeOtherService
{
    protected $userService;

    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }

    public function performComplexOperation()
    {
        // Использование логики из UserService
        $user = $this->userService->createUser([...]);
        // Другая бизнес-логика
    }
}
```

Генерация API документа

Архитектура обработки запросов LARAVEL

Последовательность обработки запроса

1. Запрос поступает на Маршрут (Route), где определяется, какой контроллер и метод должны обработать запрос.

2. **Контроллер вызывает Request**, который проверяет валидность данных согласно правилам валидации.
3. **Сервис обрабатывает данные**, выполняет основную бизнес-логику, взаимодействует с моделью, базой данных или другими внешними сервисами.
4. **Контроллер получает данные из сервиса** и передает их через Ресурс для форматирования.
5. **Ресурс возвращает ответ клиенту**, который соответствует ожидаемому формату API.

Маршрут (Route)

Маршрут определяет конечную точку API или веб-запроса, которая обрабатывается при вызове URL.

Файл: `routes/web.php` или `routes/api.php` (в зависимости от типа приложения).

Пример:

```
Route::post('/create-user', [UserController::class, 'store']);
```

Здесь при вызове маршрута `/create-user` запрос будет направлен в `UserController`, метод `store`.

Контроллер (Controller)

Контроллер обрабатывает логику запроса. Он служит связующим звеном между маршрутом и бизнес-логикой приложения, находящейся в сервисах.

Файл: `app/Http/Controllers/UserController.php`

Пример:

```
public function store(StoreUserRequest $request)
{
    // Данные, валидация пройдена
```

```
$validatedData = $request->validated();

// Логика обработки через сервис
$user = $this->userService->createUser($validatedData);

// Возвращаем обработанный ответ
return new UserResource($user);
}
```

Здесь контроллер использует запрос `StoreUserRequest` для валидации и передает данные в сервис `UserService` для дальнейшей обработки.

Запрос и правила (Request-Rules)

Запросы в Laravel представляют собой объекты, которые содержат логику валидации данных перед их передачей в контроллер. Валидация всегда происходит на этапе запросов, что делает код контроллеров чище и поддерживаемее.

Файл: `app/Http/Requests/StoreUserRequest.php`

Пример:

```
public function rules()
{
    return [
        'name' => 'required|string|max:255',
        'email' => 'required|email|unique:users,email',
        'password' => 'required|string|min:8',
    ];
}
```

Здесь запрос проверяет, что все обязательные поля (имя, email, пароль) валидны и соответствуют правилам.

Сервис (Service)

Сервисы содержат основную бизнес-логику приложения. Контроллеры взаимодействуют с сервисами для выполнения операций, таких как работа с базой данных, вызов внешних API и т.д.

Файл: `app/Services/UserService.php`

Пример:

```
public function createUser(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => Hash::make($data['password']),
    ]);
}
```

В сервисе хранится логика создания пользователя. Контроллер не занимается подробностями сохранения, он лишь вызывает сервис.

Ресурс (Resource)

Ресурсы форматируют данные для отправки обратно клиенту. Они превращают объекты моделей или коллекции в нужный JSON-формат.

Файл: `app/Http/Resources/UserResource.php`

Пример:

```
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'created_at' => $this->created_at->toDateTimeString(),
    ];
}
```

Ресурс формирует данные для ответа API, убирая ненужные поля и форматируя их в удобный вид.

Формирование сущностей в Laravel:

Пошаговое руководство

Общий путь

Процесс работы с данными в Laravel включает создание

1. Создание миграций
2. Создание моделей
3. Создание фабрик
4. Создание сервисов
5. Создание сидеров

Каждый из этих этапов выполняет свою роль в управлении данными и бизнес-логикой приложения.

MIGRATION

Создание миграции

Создайте файл миграции для таблицы, используя команду:

```
php artisan make:migration create_users_table
```

Определение структуры таблицы

В файле миграции определите структуру таблицы:

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamps();
    });
}
```

Применение миграции

Запустите миграцию для создания таблицы в базе данных:

```
php artisan migrate
```

MODEL

Создание модели

Создайте модель для вашей таблицы:

Для более эффективного создания моделей необходимо использовать библиотеку

<https://www.google.com/url?q=https://github.com/reliese/laravel&sa=D&source=docs&ust=1728309481188932&usg=A0vVaw1y6kBMVzNa1A-ztgo3FZWi>,

которая **автоматически генерирует модели** с учетом структуры базы данных и упрощает процесс описания.

Let's scaffold some of your models from your default connection:

```
php artisan code:models
```

You can scaffold a specific table like this:

```
php artisan code:models --table=users
```

You can also specify the connection:

```
php artisan code:models --connection=mysql
```

If you are using a MySQL database, you can specify which schema you want to scaffold:

```
php artisan code:models --schema=shop
```

В model строго запрещено хранить бизнес-логику.

FACTORY

Создание фабрики

Создайте фабрику для модели:

```
php artisan make:factory UserFactory --model=User
```

Определение фабрики

Определите, как будет генерироваться тестовая информация:

```
public function definition()
{
    return [
        'name' => $this->faker->name(),
        'email' => $this->faker->unique()->safeEmail(),
    ];
}
```

SERVICE

Создание сервиса

Создайте класс сервиса для управления логикой:

```
namespace App\Services;

use App\Models\User;

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

Использование сервиса

Используйте сервис в контроллере (в частности) или другом месте:

```
$userService = new UserService();
```

```
$user = $userService->create(['name' => 'John', 'email' => 'john@example.com']);
```

SEEDER

Создание сидера

Создайте сидер для заполнения таблицы тестовыми данными:

```
php artisan make:seeder UserSeeder
```

Определение сидера

Используйте фабрику для заполнения таблицы:

```
public function run()
{
    \App\Models\User::factory(10)->create();
}
```

Запуск сидера

Запустите сидер для добавления данных в базу:

```
php artisan db:seed --class=UserSeeder
```

Важные утилиты

Список будет пополняться

Генерация моделей (DEV)

<https://github.com/reliese/laravel?tab=readme-ov-file>

Файловая библиотека SPATIE

<https://spatie.be/docs/laravel-medialibrary/v11/introduction>

Работа с EXCEL документами

<https://laravel-excel.com/>

Мультиязычность AstroTomic

<https://docs.astrotomic.info/laravel-translatable>

Логирование SPATIE

<https://spatie.be/docs/laravel-activitylog/v4/introduction>

Отладка ошибок Telescope

<https://laravel.com/docs/11.x/telescope>

Локальный запуск в Docker

<https://laravel.com/docs/11.x/sail>

SOLID (изучение процесса)

SOLID – это набор принципов объектно-ориентированного проектирования, который помогает создавать более гибкие и поддерживаемые системы. Вот краткое описание каждого принципа с примерами, как они могут быть применены в Laravel, особенно в контексте паттерна Service.

Single Responsibility Principle (SRP)

Принцип единственной ответственности

Каждый класс должен иметь только одну причину для изменения. Это означает, что класс должен выполнять только одну задачу.

Пример: Создадим сервис для работы с пользователями:

```
class UserService {  
    public function createUser(array $data) {  
        // Логика создания пользователя  
    }  
  
    public function deleteUser($userId) {  
        // Логика удаления пользователя  
    }  
}
```

Open/Closed Principle (OCP)

Принцип открытости/закрытости

Классы должны быть открыты для расширения, но закрыты для модификации.

Пример: Если нам нужно добавить новую функциональность для пользователей, мы можем создать новый класс:

```
class PremiumUserService extends UserService {  
    public function createPremiumUser(array $data) {  
        // Логика создания премиум-пользователя  
    }  
}
```



```
}
```

Liskov Substitution Principle (LSP) // I thought it is something different it is about Parent And Child classes

```
//Bad practice
class Rectangle {
    private $width;
    private $height;
    public function setWidth($height){
        $this->height = $height;
    }

    public function setHeight($width){
        $this->width = $width;
    }
}
```

```
class Square extends Rectangle {

}
```

```
function clientCode(Rectangle $rec){
    $rec->setWidth(10);
    $rec->setHeight(20);
}
$rectangle = new Rectangle();
$square = new Square();
```

```
clientCode($rectangle);
```

```
clientCode($square);
```

```
//Seperating is good practice
class Rectangle {
    private $width;
    private $height;
    public function setWidth($height){
        $this->height = $height;
    }
}
```

```

        public function setHeight($width){
            $this->width = $width;
        }
    }

class Square {
    private $width;
    public function setWidth($width){
        $this->width = $width;
    }
}

function clientCodeRectangle(Rectangle $rec){
    $rec->setWidth(10);
    $rec->setHeight(20);
}

function clientCodeSquare(Square $square){
    $square->setWidth(10);
}

$rectangle = new Rectangle();
$square = new Square();

clientCode($rectangle);

clientCode($square);

```

Принцип подстановки Лисков

Объекты должны быть заменяемы их подтипами без изменения желаемых свойств программы.

Пример: Предположим, что мы используем разные сервисы для разных типов пользователей, все они должны реализовывать общий интерфейс:

```

interface UserServiceInterface {
    public function createUser(array $data);
}

class RegularUserService implements UserServiceInterface {
    public function createUser(array $data) {
        // Логика для обычного пользователя
    }
}

```

```

    }
}

class AdminUserService implements UserServiceInterface {
    public function createUser(array $data) {
        // Логика для администратора
    }
}

```

Interface Segregation Principle (ISP)

Принцип разделения интерфейсов

Клиенты не должны зависеть от интерфейсов, которые они не используют.

Пример: Разделите интерфейсы для разных типов сервисов, чтобы каждый клиент использовал только нужный интерфейс:

```

interface CreateUserInterface {
    public function createUser(array $data);
}

interface DeleteUserInterface {
    public function deleteUser($userId);
}

class UserService implements CreateUserInterface,
DeleteUserInterface {
    public function createUser(array $data) {
        // Логика создания пользователя
    }

    public function deleteUser($userId) {
        // Логика удаления пользователя
    }
}

```

Dependency Inversion Principle (DIP)

Принцип инверсии зависимостей

Зависимости должны быть абстракциями, а не конкретными классами.

Пример: Используйте инъекцию зависимостей в Laravel, чтобы внедрить сервисы:


```
class UserController extends Controller {
    protected $userService;

    public function __construct(UserServiceInterface $userService) {
        $this->userService = $userService;
    }


    public function store(Request $request) {
        $this->userService->createUser($request->all());
    }
}
```

Автоматизация и оптимизация разработки на начальном этапе (процесс описания...)

Генерация сущностей в EXCEL

1. Описание таблиц в EXCEL. Пример:  SBI AGGREGATE TABLES V2
2. Генерация миграций посредством GPT
 - а. Написание Artisan команды на создание миграции
 - б. Наполнение содержимым файл миграции
 - с. Написание Artisan команды на создание миграции
3. Генерация **моделей** на основе **таблиц**
 - а. Получить массив таблиц
 - б. На основе каждой таблицы генерировать модели с помощью библиотеки [reliese](#)
4. Генерация **сервисов** на основе **моделей**
 - а. Получить массив моделей
 - б. На основе массива генерировать сервисы в главном каталоге Services

Генерация API методов в EXCEL

1. Описание методов в EXCEL. Пример:  SBI AGGREGATE API METHODS
2. Генерация кода для **routes** посредством GPT
 - а. Генерация кода
 - б. Ручная вставка в файл routes/api.php
3. Генерация **controllers** посредством GPT
 - а. Написание Artisan команды на создание CONTROLLER файла
 - б. Вставка в контроллер методы описанные в EXCEL документе
 - i. Вставка REQUESTS
 - ii. Возврат RESOURCES
4. Генерация **requests** посредством GPT
 - а. Написание Artisan команды на создание REQUEST файла
 - б. Вставка в файл правила validation на основе excel документа
5. Генерация **services** посредством GPT
 - а. Создание в консоли команды, которая генерирует файл-класс в папке Services с суффиксом Services
 - б. Создание внутри файлов методы.

6. Генерация **resources** посредством GPT
 - а. Написание Artisan команды на создание RESOURCE файла
 - б. Вставка в контроллер массив полей, которая имеет модель
7. Формирование и наполнение фейковыми данными с помощью GPT