CET218

Advanced Web Programming

08- Laravel Concepts

Dr. Ahmed Said

Start →

6. Routing

Routes define how your application responds to specific URL requests.

Route Files

- `routes/web.php` : For browser-accessible routes. Includes middleware like session state and CSRF protection.
- `routes/api.php` : For stateless API routes. Includes `api` middleware group (throttling).
  - Routes are automatically prefixed with `/api` .

Defining Basic Routes (`routes/web.php`)

```php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\UserController; // Import Controller

// Basic GET route returning a simple view
Route::get('/', function () {
    return view('welcome'); // `resources/views/welcome.blade.php`
});

// Route mapped to a Controller method
Route::get('/users', [UserController::class, 'index']);

// Route with parameters
Route::get('/users/{id}', [UserController::class, 'show']);

// POST route for form submissions
Route::post('/users', [UserController::class, 'store']);

// Other HTTP verbs: PUT, PATCH, DELETE, OPTIONS
// Route::put('/users/{id}', [UserController::class, 'update']);
// Route::delete('/users/{id}', [UserController::class, 'destroy']);
```

- **Facade**: A facade in Laravel is a static proxy to an underlying class in the service container. The Route facade provides access to the router service. The `Route` facade is used to define routes

6. Routing (Continued)

Route Parameters

Capture segments of the URI.

```php
// Required parameter
Route::get('/posts/{id}', function (string $id) {
    return 'Post ID: ' . $id;
});

// Optional parameter with default value
Route::get('/profile/{name?}', function (string $name = 'Guest') {
    return 'Hello, ' . $name;
});

// Regular expression constraints
Route::get('/user/{id}', function (string $id) {
    // Logic...
})->where('id', '[0-9]+'); // Only numeric IDs

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

6. Routing (Continued)

Named Routes

Assign names for easy URL generation.

```php
// Define a named route
Route::get('/admin/profile', [AdminController::class, 'profile'])
     ->name('admin.profile');

// Generate URL in a Controller or View
$url = route('admin.profile');
// Output: http://your-app.com/admin/profile

// Generate URL with parameters
Route::get('/users/{id}/edit', [UserController::class, 'edit'])
     ->name('users.edit');

$url = route('users.edit', ['id' => 15]);
// Output: http://your-app.com/users/15/edit
```

- **Note:** Named routes are useful for generating URLs in views or when redirecting.
- **Example:** `route('admin.profile')` generates the URL for the named route.

6. Routing (Continued)

Route Groups

Apply middleware or prefixes to multiple routes.

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

Route::prefix('admin')->group(function () {
    Route::get('/users', [AdminUserController::class, 'index']);
});
```

- **Middleware:** Apply authentication or other middleware to a group of routes.
- **Prefix:** Add a common prefix to all routes in the group (e.g., `/admin`).

7. Controllers

Controllers group related request handling logic into a single class.

Creating Controllers

Use the Artisan command:

```bash
# Create a simple controller
php artisan make:controller PostController

# Create a resourceful controller (with CRUD methods)
php artisan make:controller PhotoController --resource

# Create an API resourceful controller (omits create/edit views)
php artisan make:controller Api/ProductController --api
```

- This creates a file like `app/Http/Controllers/PostController.php`.
- The `--resource` flag generates methods for common actions (index, create, store, show, edit, update, destroy).
- The `--api` flag generates methods for API actions (index, store, show, update, destroy) without create/edit views.

7. Controllers (Continued)

- Example Controller (`app/Http/Controllers/UserController.php`)

```php
<?php

namespace App\Http\Controllers;

use App\Models\User; // Import the User model
use Illuminate\Http\Request; // For handling request data
use Illuminate\View\View; // Type hint for view return

class UserController extends Controller
{
    // Method to display a list of users
    public function index(): View
    {
        $users = User::all(); // Fetch all users from the database
        return view('users.index', ['users' => $users]); // Pass users to the view
    }
```

7. Controllers (Continued)

- Example Controller (`app/Http/Controllers/UserController.php`)

```php
<?php
    // Method to display a single user
    public function show(string $id): View
    {
        $user = User::findOrFail($id); // Find user or throw 404
        // Pass the specific user to the 'users.show' view
        return view('users.show', ['user' => $user]);
    }

    // Method to store a new user (example for POST)
    public function store(Request $request): RedirectResponse // Example return type
    {
        // Validate request data (more on this later)
        // $validated = $request->validate([...]);

        // Create user
        // User::create($validated);

        // Redirect after creation
        // return redirect()->route('users.index');
        return redirect('/users'); // Simple redirect
```

8. Creating Your First Routes & Controllers

- Let's tie it together!
- **Goal:** Create a page to display a list of products and another for a single product.
- **Step 1: Create the Controller**

```bash
php artisan make:controller ProductController
```

  - This creates `app/Http/Controllers/ProductController.php`.

- **Step 2: Define Controller Methods**
  - Edit `app/Http/Controllers/ProductController.php`:

8. Creating Your First Routes & Controllers (Cont.)

```php
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\View\View; // Import View

class ProductController extends Controller
{
    // Display a list of products
    public function index(): View
    {
        $products = [
            ['id' => 1, 'name' => 'Laptop Pro'],
            ['id' => 2, 'name' => 'Wireless Mouse'],
            ['id' => 3, 'name' => 'Mechanical Keyboard']
        ]; // For now, let's use dummy data

        return view('products.index', ['products' => $products]); // We need a view: resources/views/products/index.blade.php
    }

    // Display a single product
    public function show(string $id): View
    {
        $product = ['id' => $id, 'name' => 'Product ' . $id, 'description' => 'Details...'];
        return view('products.show', ['product' => $product]); // // We need a view: resources/views/products/show.blade.php
```

8. Creating Your First Routes & Controllers (Cont.)

- **Step 3: Define Routes**
  - Edit `routes/web.php`:

```php
<?php

use Illuminate\Support\Facades\Route;
// Import the new controller
use App\Http\Controllers\ProductController;

Route::get('/', function () {
    return view('welcome');
});

// Route for listing all products
Route::get('/products', [ProductController::class, 'index'])
     ->name('products.index'); // Name the route

// Route for showing a single product
Route::get('/products/{id}', [ProductController::class, 'show'])
     ->name('products.show'); // Name the route
```

8. Creating Your First Routes & Controllers (Cont.)

- **Step 4: Create Views (Blade Templates)**

1. Create `resources/views/products/index.blade.php`:

```html
<h1>Product List</h1>
<ul>
    @foreach ($products as $product)
        <li>
          <a href="{{ route('products.show', ['id' => $product['id']]) }}">
            {{ $product['name'] }}
          </a>
        </li>
    @endforeach
</ul>
```

2. Create `resources/views/products/show.blade.php`:

```html
<h1>{{ $product['name'] }}</h1>
<p>ID: {{ $product['id'] }}</p>
<p>{{ $product['description'] ?? 'No description available.' }}</p>
<a href="{{ route('products.index') }}">Back to Products</a>
```

8. Creating Your First Routes & Controllers (Cont.)

- **Step 5: Test!**

  1. Run `php artisan serve`.
  2. Visit `http://127.0.0.1:8000/products`
  3. Click on a product link to visit `http://127.0.0.1:8000/products/{id}`

API Routes & Controllers

- Similar process, but use `routes/api.php`.
- Routes are automatically prefixed with `/api`.
- Controllers often return JSON instead of views.
- Typically used for frontend frameworks (React, Vue, Angular) or mobile apps.

API Routes & Controllers (Cont.)

- **Step 1: Create API Controller**

```bash
php artisan make:controller Api/ProductController --api
```

API Routes & Controllers (Cont.)

- **Step 2: Define Methods in** `Api/ProductController.php`

```php
<?php
namespace App\Http\Controllers\Api; // Note the namespace
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Http\JsonResponse; // Import JsonResponse

class ProductController extends Controller
{
    public function index(): JsonResponse {
        $products = [ /* ... dummy data ... */ ];
        return response()->json($products); // Return JSON
    }

    public function show(string $id): JsonResponse {
        $product = [ /* ... find product logic ... */ ];
        if (!$product) {
            return response()->json(['message' => 'Product not found'], 404);
        }
        return response()->json($product); // Return JSON
    }
    // store, update, destroy methods would also return JSON
}
```

Laravel Core Concepts

Diving into the Heart of the Framework (v12.x)

Today's Journey

1. **Request Lifecycle:** From Request to Response
2. **Service Container:** The Powerhouse of Dependency Injection
3. **Service Providers:** Bootstrapping Your Application
4. **Facades:** Convenient Access to Services

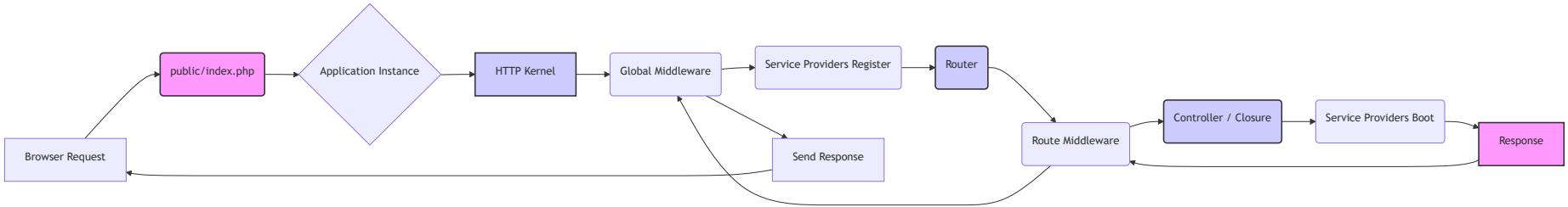
1. Request Lifecycle

Understanding how Laravel handles a request.

Request Lifecycle: Overview

- **The Journey of a Request:**
  1. **Entry Point:** Request hits `public/index.php`.
  2. **Autoloader:** Composer's autoloader loads necessary classes.
  3. **Application Instance:** Laravel application instance is created.
  4. **Kernel:** Request sent to HTTP or Console Kernel.
  5. **Bootstrappers:** Core configurations (environment, logging, etc.).
  6. **Middleware:** Request passes through global middleware stack.
  7. **Service Providers:** `register` and `boot` methods are called.
  8. **Routing:** Dispatcher sends request to a route or controller.
  9. **Route Middleware:** Request passes through route-specific middleware.

Request Lifecycle: Flow Diagram



Simplified flow diagram

Entry Point: `public/index.php`

- The gateway for all web requests.

```php
define('LARAVEL_START', microtime(true));

// Register the Composer autoloader
require __DIR__.'/../vendor/autoload.php';

// Bootstrap Laravel and handle the request
$app = require_once __DIR__.'/../bootstrap/app.php';

// Create Kernel instance and handle the incoming request
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

// Handle request, get response, send it, terminate
$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
)->send();

$kernel->terminate($request, $response);
```

- Loads Composer's autoloader.
- Retrieves the Laravel application instance from `bootstrap/app.php`.
- Retrieves the HTTP Kernel implementation.
- Calls the `handle` method on the kernel.
- Sends the response back to the browser via `send()`.
- Calls the `terminate` method (for tasks after response sent).

HTTP Kernel: `app/Http/Kernel.php`

- The central hub for incoming HTTP requests.

```php
namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel
{
    // Global HTTP middleware stack (executed on every request)
    protected $middleware = [
        \App\Http\Middleware\TrustProxies::class,
        \Illuminate\Http\Middleware\HandleCors::class,
        // ... other global middleware
    ];

    // Route middleware groups (applied to specific routes/groups)
    protected $middlewareGroups = [
        'web' => [
            \App\Http\Middleware\EncryptCookies::class,
            // ... web middleware
        ],
        'api' => [
            // ... api middleware
        ],
```

- Defines the **global middleware** stack. These run on every HTTP request.
- Defines **middleware groups** (`web` , `api`). Convenient bundles of middleware.

HTTP Kernel: `app/Http/Kernel.php` (cont.)

- The central hub for incoming HTTP requests.

```php
namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel
{
    //...

    // Route middleware (can be assigned individually to routes)
    // These can be assigned to routes individually or in groups.
    protected $middlewareAliases = [ // Renamed from $routeMiddleware in L10+
        'auth' => \App\Http\Middleware\Authenticate::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
        // ... other route middleware aliases
    ];
}
```

- Defines **middleware aliases**. Short names for individual middleware classes, assignable to routes.
- The Kernel's `handle` method orchestrates passing the request through these middleware stacks.

2. Service Container (IoC)

Laravel’s powerful tool for managing class dependencies.

What is the Service Container?

- **Inversion of Control (IoC) Container:** A mechanism for managing class dependencies and performing dependency injection.
- **Central Registry:** Holds bindings for how to resolve classes (interfaces to concrete implementations).
- **Dependency Injection:** Automatically "injects" dependencies (other objects a class needs) instead of the class creating them itself.

Why Use It?

- ✅ **Decoupling:** Reduces tight coupling between classes. Easier to swap implementations.
- ✅ **Testability:** Makes it easier to mock dependencies during testing.
- ✅ **Maintainability:** Centralized place to manage how objects are created.

Binding Basics

- Telling the container how to create an object. Usually done in a Service Provider's `register` method.
- `bind` **(Transient):** Resolves a new instance each time.

```php
// In a Service Provider's register() method
use App\Services\PaymentGateway;
use App\Services\StripeGateway;

$this->app->bind(
    PaymentGateway::class, // Abstract/Interface
    StripeGateway::class   // Concrete Implementation
);

// Resolving 'PaymentGateway' gives a new StripeGateway instance each time.
$gateway1 = app(PaymentGateway::class);
$gateway2 = app(PaymentGateway::class);
// $gateway1 !== $gateway2
```

Binding Basics (cont.)

- `singleton`: Resolves the same instance every time.

```php
// In a Service Provider's register() method
use App\Services\Logger;
use App\Services\FileLogger;

$this->app->singleton(
    Logger::class,
    FileLogger::class
);

// Resolving 'Logger' always gives the same FileLogger instance.
$logger1 = app(Logger::class);
$logger2 = app(Logger::class);
// $logger1 === $logger2
```

- `instance`: Binds an existing object instance.

```php
$service = new ExternalService(['key' => 'value']);

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

Resolution: Getting Objects Out

- How Laravel provides dependencies where they're needed.

1. `app()` Helper / `make()` Method: Manually resolve from the container.

```php
// Using the helper
$service = app(MyService::class);

// Using the Application instance
$service = $this->app->make(MyService::class);

// Using the App facade
use Illuminate\Support\Facades\App;
$service = App::make(MyService::class);
```

Less common in typical application code.

Resolution: Getting Objects Out (cont.)

2. Constructor Injection: Dependencies type-hinted in the constructor are automatically resolved.

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

    // Type-hint UserService here
    public function __construct(UserService $userService)
    {
        // Container automatically creates/injects UserService
        $this->userService = $userService;
    }

    public function show($id)
    {
        $user = $this->userService->find($id);
        // ...
    }
}
```

Most common method.

Resolution: Getting Objects Out (cont.)

3. Method Injection: Dependencies type-hinted in controller methods (or others called by the container) are resolved.

```php
class OrderController extends Controller
{
    // Type-hint Request and a custom service
    public function store(
        Request $request,
        OrderProcessor $processor,
        int $id // Route parameters resolved too!
    )
    {
        // Container injects Request instance and OrderProcessor
        $order = $processor->create($request->all());
        // ...
    }
}
```

Useful for dependencies only needed in one method.

Advanced Container Features

- **Contextual Binding:** Define different implementations based on where a dependency is being injected.

```php
// When PhotoController needs Storage, use S3Storage
$this->app->when(PhotoController::class)
          ->needs(Storage::class)
          ->give(S3Storage::class);

// When VideoController needs Storage, use LocalStorage
$this->app->when(VideoController::class)
          ->needs(Storage::class)
          ->give(LocalStorage::class);
```

Advanced Container Features (cont.)

- **Tagging:** Group related bindings together. Useful for collections of services (e.g., report types, payment drivers).

```php
$this->app->bind(StripeGateway::class, fn() => ...);
$this->app->bind(PayPalGateway::class, fn() => ...);

$this->app->tag([StripeGateway::class, PayPalGateway::class], 'payment.gateways');

// Resolve all tagged instances
$gateways = $this->app->tagged('payment.gateways'); // Returns an array/iterable
```

- **Extending Bindings:** Modify or decorate resolved services.

```php
$this->app->extend(SomeService::class, function ($service, $app) {
    // Add logging or modify the original service
    return new DecoratedService($service);
});
```

3. Service Providers

The central place to configure and bootstrap your application.

Role of Service Providers

- **Bootstrapping:** The primary way to "boot up" parts of your application.
- **Configuration:** Registering service container bindings, event listeners, middleware, routes, etc.
- **Organization:** Group related bootstrapping logic together.

- All service providers extend `Illuminate\Support\ServiceProvider`.
- They live in the `app/Providers` directory.
- Laravel includes several core providers, and you can generate your own provider using `php artisan make:provider RiakServiceProvider`.

Service Provider Structure

- Two key methods: `register()` and `boot()`.

```php
namespace App\Providers;
use Illuminate\Support\ServiceProvider;
use App\Services\Riak\Connection as RiakConnection; // Example Service
class RiakServiceProvider extends ServiceProvider {
    /**
     * Register any application services.
     * Called BEFORE boot(). ONLY bind things into the container here.
     * DO NOT try to use services registered here yet.
     */
    public function register(): void {
        $this->app->singleton(RiakConnection::class, function ($app) {
            return new RiakConnection(config('riak')); // Example binding
        });
    }
    // ...
}
```

- `register()` **Method:**
  - **Purpose:** Only bind things into the service container (`bind` , `singleton` , `instance`).

Service Provider Structure

- Two key methods: `register()` and `boot()`.

```php
// ...
class RiakServiceProvider extends ServiceProvider {
    // ...
    /** Bootstrap any application services.
     * Called AFTER all providers have been registered.
     * Safe to use other registered services here.
     * Good for: registering event listeners, publishing assets, adding routes.
     */
    public function boot(): void {
        // Example: Publishing configuration file
        // $this->publishes([__DIR__.'/../config/riak.php' => config_path('riak.php'),
        // ]);
    }
}
```

- `boot()` **Method:**
  - **Purpose:** Access other services, register event listeners, include route files, publish assets, etc.
  - **Timing:** Called after all service providers have executed their `register` methods. It's safe to use any registered service here.

Registering Providers

Your custom service providers need to be registered in `config/app.php`.

```php
// config/app.php

'providers' => ServiceProvider::defaultProviders()->merge([
    /* Package Service Providers... */

    /* Application Service Providers... */
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

    // Add your custom provider here!
    App\Providers\RiakServiceProvider::class, // <--- REGISTERED

])->toArray(),
```

- Add your provider's class name to the `providers` array.
- Providers are registered (their `register` method called) in the order they appear.
- Then, all `boot` methods are called.

Deferred Providers

Optimize performance by only loading a provider when its services are actually needed.

- **How:** Implement the `DeferrableProvider` interface and add a `provides()` method.
- `provides()` **Method:** Returns an array of the service container bindings registered by the provider.

```php
namespace App\Providers;

use Illuminate\Contracts\Support\DeferrableProvider; // 1. Implement interface
use Illuminate\Support\ServiceProvider;
use App\Services\Riak\Connection as RiakConnection;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider // 1. Implement interface
{
    public function register(): void { /* ... binding ... */ }

    /** Get the services provided by the provider.
     * Tells Laravel which bindings this provider handles.
     */
    public function provides(): array // 2. Add provides() method
    {
        return [RiakConnection::class]; // 3. List bindings
    }
}
```

4. Facades

Providing a "static" interface to services in the container.

What are Facades?

- **Static Proxy:** Facades provide a convenient, memorable, "static" syntax for accessing services bound in the IoC container.
- **Under the Hood:** They are not truly static methods in the traditional sense. They resolve an object instance from the container and call the method on that object.

Facade Usage Example:

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

// Using the Cache facade
$value = Cache::get('my-key');

// Using the Route facade
Route::get('/users', [UserController::class, 'index']);
```

Looks static, but isn't!

Equivalent Container Resolution:

```php
use Illuminate\Contracts\Cache\Factory as CacheFactory;
use Illuminate\Routing\Router;

// Manually resolving from container
$cache = app(CacheFactory::class); // or 'cache' alias
$value = $cache->get('my-key');

$router = app(Router::class); // or 'router' alias
$router->get('/users', [UserController::class, 'index']);
```

This is what facades do internally.

What are Facades? (cont.)

Benefits:

- ✅ **Conciseness & Readability:** Often shorter and more expressive than injecting or resolving manually.
- ✅ **Memorability:** Easy-to-remember static-like calls.
- ✅ **Testability:** Laravel provides excellent Facade mocking capabilities.

How Facades Work

1. You call a "static" method: `Cache::get('key')`.
2. PHP's `__callStatic()` magic method on the base `Facade` class intercepts the call.
3. The facade determines the **service container binding name** (e.g., `'cache'`) via its `getFacadeAccessor()` method.
4. It resolves the underlying service instance from the container: `app('cache')`.
5. It calls the actual instance method on the resolved object: `$resolvedInstance->get('key')`.

How Facades Work (cont.)

```php
// Example: Illuminate\Support\Facades\Cache

namespace Illuminate\Support\Facades;

class Cache extends Facade
{
    /**
     * Get the registered name of the component.
     * This tells the Facade *which* service to resolve from the container.
     *
     * @return string
     */
    protected static function getFacadeAccessor(): string
    {
        return 'cache'; // The binding name in the container
    }
}

// When you call Cache::get('foo'), it effectively does:
// 1. Call Cache::getFacadeAccessor() -> returns 'cache'
// 2. Resolve 'cache' from container -> app('cache') -> returns CacheManager instance
// 3. Call 'get' on the CacheManager instance -> app('cache')->get('foo')
```

Creating Your Own Facades

1. The Service Class: Your underlying service.

```php
// app/Services/PaymentGateway.php
namespace App\Services;

class PaymentGateway
{
    public function process(float $amount)
    {
        // Process payment...
        return true;
    }
}
```

Creating Your Own Facades (cont.)

2. The Facade Class: Extends `Illuminate\Support\Facades\Facade` and implements `getFacadeAccessor`.

```php
// app/Facades/Payment.php
namespace App\Facades;

use Illuminate\Support\Facades\Facade;

class Payment extends Facade
{
    protected static function getFacadeAccessor(): string
    {
        // Return the container binding name
        return \App\Services\PaymentGateway::class;
        // Or a custom string alias like 'payment.gateway'
    }
}
```

Creating Your Own Facades (cont.)

3. Binding & Alias (Optional): Ensure the service is bound in the container (e.g., in a Service Provider). Optionally add an alias in `config/app.php`.

```php
// In AppServiceProvider::register()
$this->app->singleton(
    \App\Services\PaymentGateway::class,
    fn() => new \App\Services\PaymentGateway()
);

// In config/app.php (optional, for convenience)
'aliases' => Facade::defaultAliases()->merge([
    'Payment' => App\Facades\Payment::class, // Add alias
])->toArray(),

// Usage:
use App\Facades\Payment; // or use Payment; if aliased
Payment::process(100.00);
```

Facades: Considerations

- **"Static" Nature:** While convenient, heavy use can sometimes obscure where dependencies come from compared to explicit constructor injection.
- **Scope Creep:** Easy to call facades from anywhere, potentially leading to less organized code if not used thoughtfully (e.g., calling `DB::` directly in a Blade view is generally discouraged).
- **Testing:** Laravel's facade mocking (`Cache::shouldReceive(...)`) is powerful but works differently than mocking regular objects.

Facades: Best Practices

Facade vs. Dependency Injection:

- **Facades:** Great for brevity, common services (`Cache`, `Log`, `Route`), and in places where injection is awkward (e.g., sometimes within service providers, configuration files).
- **Injection:** Generally preferred for core application logic (controllers, services) as it makes dependencies explicit and clear. Often leads to more testable and decoupled code.
- **Real-time Facades:** Prefix your service's namespace with `Facades\` without creating a dedicated Facade class (e.g., `use Facades\App\Services\PaymentGateway;`). Laravel generates them on the fly. Convenient but less explicit.