

# Laravel Advance

## Request Life cycle:

### Request Entry

- The user makes an HTTP request to the Laravel application.
- The request is handled by the server (e.g., Apache or Nginx) and routed to `public/index.php`.

### Autoloading

- Composer's autoload file (`vendor/autoload.php`) is loaded to handle class autoloading.

### Bootstrap Application

- `bootstrap/app.php` is loaded to create the Laravel application instance.
- An instance of `Illuminate\Foundation\Application` is initialized.

### HTTP Kernel Initialization

- The application resolves the **HTTP Kernel** (`app/Http/Kernel.php`) to handle the request.
- The kernel defines global middleware, route middleware, and middleware groups.

## Register Service Providers

- Service providers listed in `config/app.php` are registered.
- These providers boot services, bindings, and dependencies for the application.

## Middleware Processing (Before Request)

- Global middleware is applied (e.g., session handling, encryption, etc.).
- Route-specific middleware is executed for additional functionality.

## Routing

- The `RouteServiceProvider` maps the request to a route defined in `routes/web.php` or `routes/api.php`.
- The route resolves the associated controller, closure, or resource method.

## Controller Execution

- The matched controller or closure is executed.
- Dependencies for the controller are injected by the **Service Container**.

## Business Logic and Database Interaction

- Models, Eloquent ORM, or Query Builder interact with the database.
- Any required data processing or business logic is executed.

## View Rendering

- The controller may return a view, which is processed by the Blade templating engine.
- The Blade templates are compiled into plain PHP for efficiency.

## Response Creation

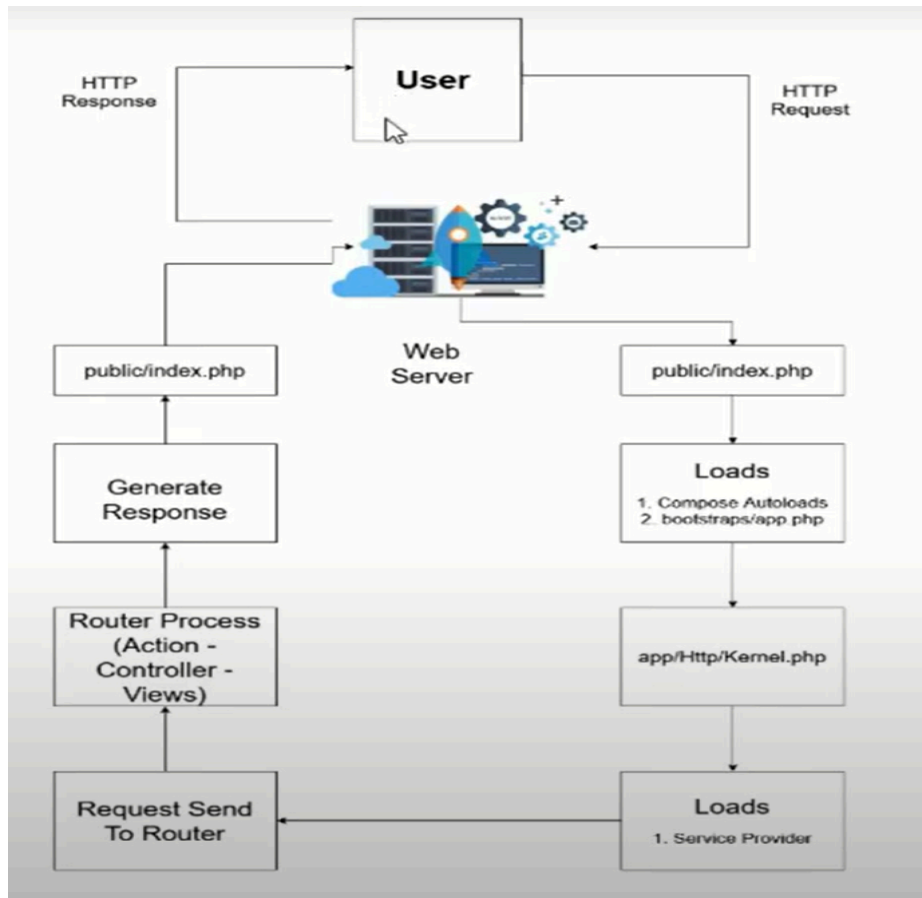
- A response object (`Illuminate\Http\Response`) is created to encapsulate the output.
- The response may include views, JSON, redirects, or files.

## Post-Middleware Processing

- Middleware with `terminate()` methods are executed after the response is sent (e.g., logging, cleaning up resources).

## Request Completion

- The application lifecycle ends, and the response is delivered to the client.



## Dependency Injection:

Dependency Injection (DI) is a technique where **a class depends on another class** but instead of creating an instance inside the class, it gets "injected" from the outside.

Dependency Injection (DI) is a design pattern used to remove hard coded dependencies in a class, making the code **more flexible, reusable, and easier to test**.

In simple terms:

- Instead of a class creating its own dependencies, **they are "injected" from the outside**.
- This allows for easy swapping of different implementations without modifying the class.

There are number of ways to Inject Dependency –

1. Constructor Injection
2. Setter Injection
3. Interface Injection

## Constructor Injection

PHP

```
Class Client {  
    public $service;  
    public function __construct(CarService $service){  
        $this->service = $service;  
    }  
}
```

## Setter Injection

PHP

```
Class Client {  
    public $service;  
    public function setService(CarService $service){  
        $this->service = $service;  
    }  
}
```

## Interface injection

PHP

```
interface Services{  
    public function setService(CarService $service);  
}  
Class Client implements Services {  
    private $service;  
    public function setService(CarService $service){  
        $this->service = $service;  
    }  
}
```

Copy

# Service Container:

Service Container or IoC in laravel is responsible for managing class dependencies meaning not every file needs to be injected in class manually but is done by the

Service Container automatically. Service Container is mainly used in injecting classes in controllers like Request object is injected. We can also inject a Model based on id in route binding.

For example, a route like below:

```
Route::get('/profile/{id}', 'UserController@profile');
```

With the controller like below

```
public function profile(Request $request, User $id)

{

    //

}
```

In the UserController profile method, the reason we can get the User model as a parameter is because of the Service Container as the IoC resolves all the dependencies

in all the controllers while booting the server. This process is also called route-model binding

## Service Provider:

A Service Provider is a way to bootstrap or register services, events, etc before booting the application. Laravel's own bootstrapping happens using Service Providers as well. Additionally, it registers service container bindings, event listeners, middlewares, and even routes using its service providers.

If we are creating our application, we can register our facades in provider classes.

## Type of service providers: (Before laravel 11)

When we install a fresh laravel project, it comes with 5 service provider classes, all provider classes have a common boot method, app service provider has default register method in it, we can add register method in other provider classes too.

- App service provider
- Auth service provider
- Broadcast service provider
- Event service provider
- Route service provider



## Difference of boot and register method:

### Register:

- **Purpose:** Used to bind services into the service container.
- **When it runs:** This method is called **before** the application is fully booted.
- **What to do here:**
  - Register bindings (e.g., interfaces to concrete classes).
  - Register singletons, aliases, or service providers.
- **Limitations:**
  - Avoid using other services or dependencies here because the application is not fully booted yet.
  - For example, you cannot access the Request object or other bound services in the register method.

Example:

```
php                                                                    Copy

public function register()
{
    $this->app->bind(PaymentGateway::class, StripePaymentGateway::class);
}
```

### Boot:

- **Purpose:** Used to perform actions after all services have been registered.
- **When it runs:** This method is called **after** all service providers have been registered.
- **What to do here:**
  - Register event listeners.
  - Define routes or view composers.
  - Perform any actions that depend on other services being registered.
- **Advantages:**

- You can access other services and dependencies here because the application is fully booted.

**Example:**

```
php Copy  
  
public function boot()  
{  
    Event::listen(UserRegistered::class, SendWelcomeEmail::class);  
}
```

## App Service Provider:

App service provider is used for changes which can be used in whole application, it allows you register your own services throughout the laravel application, for example if you don't want to use default laravel pagination view, you can change its view in the app service provider's boot method, similarly if you want to make your own blade directive such as (@date) so you can make it in the app service provider's boot method.

## Auth Service Provider:

Auth service provider is used to register policies and change in authentication functionality.

## Broadcast Service Provider:

## Event Service Provider:

Event service provider is used to manage events in application, In simple terms, the

EventServiceProvider helps you manage and respond to events that occur within your Laravel application.

## Route Service Provider:

When you have to a lot of routes and you want to separate them in another file, you can do this in route service provider class, we can also write rules for route parameters in the boot method, we can change resource route urls in the url bar, we can define route model binding etc.

## Difference of all providers:

Only difference between all these provider classes is their names. We can bind events in AppServiceProvider too, but it is good practice to write all the logic in its corresponding class.

## Facades:

Facades in Laravel are simple static-like shortcuts to access Laravel's services (like Cache, DB, Auth) without needing to manually inject dependencies.

Facades are globally accessible throughout your application without needing to inject dependencies into constructors or methods.

## Why Use Facades?

**Easy to Use** – No need to create objects manually

**Clean Code** – No need for dependency injection in small tasks

**Readable** – Looks like a static method but works with Laravel's service container

Laravel provides some default facades too which we can use like in the pic below:

# Laravel facades

*vendors > laravel > framework > src > illuminate > support > Facades*

## Some Default Facades....

Auth::user()

DB::table('users')

Cache::put()

Log::info('logging')

Route::get()

View::make('home');

## [When To Use Facades](#)

Facades have many benefits. They provide a terse, memorable syntax that allows you to use Laravel's features without remembering long class names that must be injected or configured manually. Furthermore, because of their unique usage of PHP's dynamic methods, they are easy to test.

We can make our own custom facades as well, in order to make our own facade of any class we will create a class first, For example if we are making a facade of the **Invoice class**,

```
namespace App\Facades;

class Invoice {

    public function CompanyName() {
        return "Company name is ooober";
    }

}
```

we will make a class named **InvoiceFacade**,

```
namespace App\Facades;
use Illuminate\Support\Facade;

class InvoiceFacade extends Facade {
|
}
```

In this class we will make a private method named **getFacadeAccessor()** and we return our facade name from this method, we will use this name for our facade throughout the app.

```
namespace App\Facades;
use Illuminate\Support\Facade;

class InvoiceFacade extends Facade {
|
|    protected static function getFacadeAccessor() {
|        return "Invoice"
|    }
|
}
```

Now we will bind **Invoice** class with name **Invoice** in the **AppServiceProvider**, like in the pic below

```

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Facades\Invoice;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->bind('Invoice', function($app){
            return new Invoice();
        });
    }
}

```

After binding, we will create its alias in the **Config\app.php** like in the pic below

```

'aliases' => Facade::defaultAliases()->merge([
    // 'ExampleClass' => App\Example\ExampleClass::class,
    'Invoice' => App\Facades\InvoiceFacade::class
])->toArray(),

```

Now we can use facade like this,

```

Route::get('/', function () {
    echo Invoice::companyName();

    // return View::make('welcome');
});

```

# Collections:

The Laravel collection is a useful feature of the Laravel framework. A collection works like a PHP array, but it is more convenient. The collection class is located in the `Illuminate\Support\Collection` location. A collection allows you to create a chain of methods to map or reduce arrays.

We can create collections with the `collect()` helper function.

```
$collection = collect([1, 2, 3]);
```

We can use multiple methods on the collections which are given by laravel, for example, **`sum()`**, **`avg()`**, **`all()`**, **`sort()`**,**`map()`**,**`filter()`** etc.

## `map()`:

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:



```
$collection = collect([1, 2, 3, 4, 5]);

$multipled = $collection->map(function (int $item, int $key) {
    return $item * 2;
});

$multipled->all();

// [2, 4, 6, 8, 10]
```

## filter():

The `filter` method filters the collection using the given callback, keeping only those items that pass a given truth test:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function (int $value, int $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to `false` will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);  
  
$collection->filter()->all();  
  
// [1, 2, 3]
```

## Contracts:

Laravel's Contracts are a set of interfaces that define the core's provided by the framework. For example, a Queue contract defines the methods needed for queueing jobs, while the Mailer contract defines the methods needed for sending email.

Laravel contracts contain interfaces about related functionality.

## Queues:

Laravel provides this feature to make queue for different events, in simple words if you are performing a task which takes time to complete, you can queue it, let's say user is adding data into database, data is big and it will take time to upload, so we will make queue for it, benefit of the queue is, it doesn't block user, for example if user uploading a big amount of data which takes time so it

will keep uploading data in the background meanwhile user can perform other action in parallel.

## How to use queues:

First we will make a queue table in the database using artisan command.

```
php artisan queue:table  
php artisan migrate
```

After making the database table we will make a job class through artisan command.

```
php artisan make:job ProcessPodcast
```

Change QUEUE\_DRIVER value to database in .env file

Once a job is created, the job class will contain a handle method in which we add all our logic which we want to queue.

```
<?php
```

```
namespace App\Jobs;  
  
use App\AudioProcessor;  
use App\Podcast;  
use Illuminate\Bus\Queueable;  
use Illuminate\Contracts\Queue\ShouldQueue;
```

```

use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable,
    SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param AudioProcessor $processor
     * @return void
     */
    public function handle(AudioProcessor
$processor)
    {
        // Process uploaded podcast...
    }
}

```

```
}
```

After all these things, we will have to trigger the job like below

```
ProcessPodcast::dispatch($podcast);
```

If you want to delay your queue the you will use delay method like below,

```
ProcessPodcast::dispatch($podcast)
```

```
->delay(now()->addMinutes(10));
```

This will not stop user from performing any action and it will start queue job after ten minutes,

After all these change just run the queue worker to run the queue

```
php artisan queue:work
```

## Difference of queues and jobs:

### Queues

#### 1. Definition:

- A **queue** is a data structure that follows the **First-In-First-Out (FIFO)** principle. It holds tasks (jobs) that need to be processed asynchronously.
- In the context of frameworks like Laravel, a queue system allows you to defer the processing of tasks to a later time or to a separate process.

#### 2. Purpose:

- Queues are used to manage and distribute tasks efficiently, especially for time-consuming operations like sending emails, processing images, or handling API requests.
  - They help improve application performance by offloading tasks to background workers.
3. **How It Works:**
- Tasks (jobs) are added to a queue.
  - A queue worker (or multiple workers) processes the jobs in the queue one by one.
4. **Example:**
- In Laravel, you can configure queues to use different backends like Redis, database, or Amazon SQS.
  - Example: Sending 1,000 emails. Instead of sending them synchronously (which would block the application), you add them to a queue, and a worker processes them in the background.

## Jobs

1. **Definition:**
- A **job** is a unit of work that needs to be performed. It encapsulates the logic for a specific task.
  - In Laravel, jobs are typically classes that implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating they should be processed asynchronously.
2. **Purpose:**
- Jobs are used to define the specific task that needs to be executed, such as sending an email, generating a report, or processing data.
  - They are the individual units of work that are pushed into a queue.
3. **How It Works:**
- A job is created and dispatched to a queue.
  - The queue worker picks up the job and executes its `handle` method.
4. **Example:**
- In Laravel, you might create a job called `SendWelcomeEmail` that sends an email to a user.
  - Example code:

```
class SendWelcomeEmail implements ShouldQueue
{
    public function handle()
    {
        // Logic to send the email
    }
}
```

You dispatch the job to a queue:

```
SendWelcomeEmail::dispatch($user);
```

## Event Listeners and Events:

### Events:

We can use events when we want to perform any action on a specific event. For example if a user changes his profile picture and we want to send him email after the completion then we can use events, or if we want to send slack notification on specific action, we can use events for that.

We make events by the artisan command, let's make a post event.

Php artisan make:event PostEvent

After running this command a new event will be created in the App\Events directory.

## Listeners:

Listeners are the part of events, when we create an event, we need to make the listener listen to that event and put all our logic in the events.

Same like events, we can make listeners through the artisan command,

```
Php artisan make:listener NotifyUser
```

After running this command a new listener class will be created in the App\Listeners directory.

## Why Use Events & Listeners?

1. **Separation of Concerns** → Keeps logic clean by separating event triggers from responses.
2. **Improves Maintainability** → Changes to one part of the app don't require changes to others.
3. **Better Performance** → Offloads time-consuming tasks to the background using queues.
4. **Extensibility** → Easily add more listeners without modifying existing code.



## **Notifications:**

Notifications can be used to notify users about anything, for example we have a functionality of events and listeners, when a user signed up, it triggers an event which sends notification to the admin about the newly signed up user. We can send notification to the listener of that event.

## **Cron job & task scheduling:**

### **Task Scheduling:**

If we want to do a task at a later point of time, you will schedule it at a certain point of time. For this we use laravel task scheduling or cron jobs.

### **Cron Job:**

In the server we schedule a task using cron jobs.

To perform a laravel cron job, we will write all the logic in the custom artisan command class and then we schedule that command into `App\Console\Kernel.php` class.

```

namespace App\Console;

use ...

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        //
    ];

    /**
     * Define the application's command schedule.
     *
     * @param \Illuminate\Console\Scheduling\Schedule $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        // $schedule->command('inspire')->hourly();
    }

    /**
     * Register the commands for the application.
     *
     * @return void
     */
    protected function commands()
    {
        $this->load(paths: __DIR__.'/Commands');

        require base_path(path: 'routes/console.php');
    }
}

```

## Eager Loading:

When we load relations along with the model it is called eager loading, see example below:


```
$heading_attributes = Family::with(['attributes'])->get();
```

In the above example, we are loading the Family model with its relation with attributes. We use eager loading in fetching large amounts of data.

## Lazy Loading:

Lazy loading in Laravel is a feature that allows you to load related data on-demand, rather than loading it all at once when retrieving a record from the database. This can help to improve the performance of your application by reducing the number of database queries executed.

php

 Copy code

```
$post = Post::find(1);

// Lazy load the comments
$comments = $post->comments;

// Access each comment
foreach ($comments as $comment) {
    echo $comment->text;
}
```

# Laravel Relations:

Laravel has different relations

1. One to One
2. One to Many
3. Many to Many
4. Has one Through
5. Has many Through
6. One to One (polymorphic)
7. One to Many (polymorphic)
8. Many to Many (polymorphic)

## One to One:

In this relationship, one database record will be related to the other only one record, for example we have two database tables, User and Phones, now we have a scenario in which a user can only have one phone, so that we can say this is a one to one relationship between user and posts.

## One to Many:

In this relationship, one database record could be related to one or more records, for example we have two tables named Users and Posts, so now in this

case if a user can add more than one post, let's suppose if a user adds 2 posts, so that those 2 posts will be related to that particular user. In this case we can say we have one to many relationship.

## Many to Many:

In this relationship, multiple database records could be related to other multiple records, let's suppose we have 2 tables, Users and Roles, so if a user have roles of author, publisher, these two roles can also be assigned to other users as well, for this purpose we have to make three database tables,

- **Users**
- **Roles**
- **Roles\_user**

Roles\_user is a pivot table in which we will save user\_id and role\_id as a foreign key. When we call a relationship from any model whether it is user table or role table, it will load the relationship from pivot table automatically.

**Pivot table name should be in alphabetically ascending order.**

Has one through:

In this relationship type we can access third tables records from the database without making relationship, let's suppose we have a car maintenance application in which we have three tables:

- **cars**
- **owners**
- **mechanics**

In these tables, cars table will have column mechanic\_id and owners table will have car\_id as foreign key, now we can access owners table record through cars table.

Has many through:

This is the convenient way to accessing relations, for example we have 3 tables

- **countries**
- **users**
- **posts**

Now in the post table we don't have country\_id as a foreign key, we have user\_id instead, and users table have

country\_id as a foreign key, now we can get posts of the user by calling its relationship into the countries model.

## One to One (polymorphic) :

A one-to-one polymorphic relationship is a situation where one model can belong to more than one type of model but on only one association. For example we have two tables, Users and Posts, and they both could have images, so we will create a third table name images to save images for the both of the tables, see example below:

posts

id - integer

name - string

users id - integer

name - string

images

id - integer

url - string  
imageable\_id - integer  
imageable\_type - string

Take note of the `imageable_id` and `imageable_type` columns on the `images` table. The `imageable_id` column will contain the ID value of the post or user, while the `imageable_type` column will contain the class name of the parent model. The `imageable_type` column is used by Eloquent to determine which "type" of parent model to return when accessing the `imageable` relation

## One to Many (polymorphic):

A one-to-many polymorphic relation is similar to a simple one-to-many relation; however, the target model can belong to more than one type of model on a single association.

```
posts
  id - integer
  title - string
  body - text
```

```
videos
  id - integer
  title - string
```



```
url - string

comments
  id - integer
  body - text
  commentable_id - integer
  commentable_type - string
```

In the above table structures, a post or video can have multiple comments.

## Template Engine:

Laravel uses blade as template engine.

## Eloquent:

Eloquent is the ORM used to interact with the database using Model classes. It gives handy methods on class objects to make a query on the database.

## Throttling:

Throttling is the function in Laravel through which

we can limit incoming requests. For throttling, Laravel provides a middleware that can be applied to routes and it can be added to the global middlewares list as well to execute that middleware for each request.

## Query Scope:

There are two types of scopes in laravel,

- Local scope
- Global scope

### Local scope:

You can define local scope if you want to implement the same query for many times, in this way you will only have to write a query for once in a function and you will use that function in multiple classes.

### Global scope:

Global scope can be used when you want a query run every time when the class runs, we will create this scope in the booted method of the class.

## Traits:

Php doesn't have multiple inheritance, so that's why we use traits for that.

Laravel follows a different architectural pattern called "Composition over Inheritance" to promote code reuse and maintainability. It encourages developers to use traits, interfaces, and composition to achieve code organization and reuse, rather than relying heavily on class inheritance.

## Create()/Insert():

These both used to insert data into database,

### Create:

Laravel eloquent uses create() method to save data into the database, if you have registered various laravel events like creating, updating etc in the model this function runs all laravel events.

## Insert:

Laravel query builder uses `insert()` method to save data into the database, it will not run any laravel event.

## Gates:

Gates are the functions defined in the **AuthServiceProvider.php** class. These are the functions in which we can check whether this logged in user can perform this action or not, these can be used anywhere in the project.

## Policies:

Policies are similar to the gates, only difference between them is, gates can only be defined in **AuthServiceProvider.php** class, but policies are the class of specific model class. For example if we want to make actions for the user model so we can make policy for it.

## Guards:

Guards differentiate authenticated users, whether it is admin or simple user. It will differentiate between their sessions or cookies too. Laravel uses session guard to

authenticate the user and if we are making apis then laravel uses token guard. Every guard has its own separate session, it means we can login as an admin and user in the same browser.

## Cascade:

Cascade is used to update and delete data from both parents and child tables. The keyword CASCADE is used as a conjunction while writing the query of ON DELETE or ON UPDATE.

## Providers:

Providers define how users will be retrieved from the database, laravel provides two ways to retrieve users from storage,

- Eloquent
- Query Builder

We can choose either of these options in the config/**auth.php** file.

## Observers:

We use observers when we want to trigger an event on action of any model, for example we have a user model and we want to perform an action when a new user is created, we will create an observer class which listens the created event when a new user is created and perform a desired task, for example we want to send an email to every new user, we can do this through observer. Once the observer class is created, we need to bind it to the user model in the EventServiceProvider class.

## Route Model Binding:

Route model binding is a feature in Laravel that allows you to bind a route parameter to a model instance. This makes it possible to retrieve a model instance directly from the route parameters, without having to fetch it from the database yourself. For example we want product model binding, we will do this like in the example below:

```
public function show(Product $product)
{
    return view('products.show', compact('product'));
}
```

We just need to pass id of the product in the route, and laravel will get its object from the database automatically,

## Replacing id with another field:

If we want to get record by the slug instead of id we can also do that by the multiple ways, see example below:

```
Route::get( uri: 'products/{product:slug}/export', [\App\Http\Controllers\ProductController::class, 'export']);
```


Or we can make its function instead of passing it with parameter, like below:

```
public function getRouteKeyName()  
{  
    return 'slug';  
}
```

## Fillable:

The `fillable` property is an array that lists the attributes that should be mass-assignable. For example:

php

 Copy code


```
class User extends Model  
{  
    protected $fillable = [  
        'name',  
        'email',  
        'password',  
    ];  
}
```

In this example, only the `name`, `email`, and `password` attributes can be mass-assigned when creating or updating a model instance. All other attributes will be ignored.

# Guarded:

The ``$guarded`` property is an array that lists the attributes that should be protected from mass-assignment. For example:

php

 Copy code

```
class User extends Model
{
    protected $guarded = [
        'id',
        'created_at',
        'updated_at',
    ];
}
```

In this example, the ``id``, ``created_at``, and ``updated_at`` attributes are protected from mass-assignment. When creating or updating a model instance, these attributes will not be updated.

# Macros:

In Laravel, macros are a way to extend the functionality of built-in classes or add custom methods to classes. They allow you to define reusable pieces of code that can be called like any other method.

## How to write macros:

Let's suppose we want to write a macro which gives us email of the authenticated user, we will make macro in



the register method of the app service provider or we can make also make our own service provider for that like in the example below:

```
php Copy code

use Illuminate\Support\ServiceProvider;
use Illuminate\Http\Request;

class RequestMacroServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        Request::macro('userEmail', function () {
            $user = $this->user();

            if ($user) {
                return $user->email;
            }

            return null;
        });
    }
}
```

Now you can call this method anywhere in the app like below:

```
Request::userEmail();
```

# Form Request:

Laravel provides a class to validate request separately instead of writing whole validation logic into the controller itself, it will make our code cleaner and much more readable, form request can be made by the artisan command it has two methods by default, `authorize` and `rules`, we can add authorization logic into the `authorize` method or we can write rules in the `rules` method, later we can type hint it in the controller method to use, like in the example below:

```
php artisan make:request UserStoreRequest
```

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'email' => 'required|email|unique:users',
        'name' => 'required|string|max:50',
        'password' => 'required'
    ];
}
```

```
class UserController extends Controller
{
    public function store(UserStoreRequest $request)
    {
        // Will return only validated data

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

## Design Patterns:

Design patterns are reusable solutions to common problems that occur in software design. They are like templates that can be adapted to fit different situations, making the design process more efficient and effective.

In simple terms, a design pattern is a way of thinking about and approaching a problem that has been proven to be successful in the past. By using a design pattern, you can save time and avoid re-inventing the wheel, because the pattern provides a tested and proven solution to a common problem.

For example, consider the problem of allowing a user to add items to a shopping cart on an e-commerce website. A common design pattern for solving this problem is the Model-View-Controller (MVC) pattern. This pattern separates the data (the model), the user interface (the view), and the control logic (the controller) into separate components, making it easier to maintain and extend the software over time.

In summary, design patterns provide a structured and efficient approach to solving common problems in software design. They can help you write better, more maintainable code and speed up the development process.

# Types of laravel design patterns:

1. Singleton: Laravel uses the Singleton pattern to ensure that only one instance of a class is created and used throughout the application.
2. Facade: Laravel's Facade pattern provides a simple and easy-to-use interface for accessing complex subsystems, allowing you to use these subsystems without needing to know the details of how they work.
3. Factory: Laravel's Factory pattern is used to generate instances of classes and objects dynamically, based on the input provided.
4. Strategy: Laravel's Strategy pattern allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. This pattern is useful for defining a set of interchangeable algorithms that can be used to achieve the same goal in different ways.
5. Repository: Laravel's Repository pattern provides a way to separate the logic that retrieves data from the data storage layer in your application, making it easier to manage data access and decoupling it from the business logic.
6. Observer: Laravel's Observer pattern allows you to define a one-to-many relationship between objects, where a change in one object triggers a notification to a set of dependent objects.

## Singleton Design Pattern:

The Singleton design pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. This pattern is useful when you want to restrict the instantiation of a class to a single object, and you need to have a single point of access to that object throughout your application.

# Repository Design Pattern:

This design pattern separates business logic and database related operations, like if we have written all the code in the controller, we will shift the database related code to the repository class and business logic will remain in the controller, see the example below:

Here's a step-by-step explanation of how to implement it:

1. **Interface:** First, you create an interface that describes the methods that will be used to access the data. For example, if you're working with a User object, your interface might include methods like `getAllUsers()`, `getUserById($id)`, `deleteUser($id)`, and so on.
2. **Repository:** Then, you create a repository class that implements the interface. This class contains the actual database queries.
3. **Service Provider:** You bind the interface to its implementation in a service provider. This is typically done in the `register` method of a service provider.
4. **Controller:** Finally, you inject the interface (not the concrete class) into your controller. This can be done through the constructor or a method, and Laravel's service container will automatically resolve the correct implementation.

```
// The Interface
interface UserRepositoryInterface {
    public function getAllUsers();
}

// The Repository
class UserRepository implements UserRepositoryInterface {
    public function getAllUsers() {
        return User::all();
    }
}
```

In the above example we have created an interface and repository class, and defined getAllUsers() method in interface and implements it in the repository.

```
// Service Provider
public function register()
{
    $this->app->bind(
        'App\Repositories\UserRepositoryInterface',
        'App\Repositories\UserRepository'
    );
}
```

Now we have bound the interface to the repository class, so that we can access all the functionality written in the repository class through the interface in the controller.

```
// The Controller
class UserController extends Controller {

    protected $users;

    public function __construct(UserRepositoryInterface $users)
    {
        $this->users = $users;
    }

    public function index()
    {
        return $this->users->getAllUsers();
    }
}
```

Now in the above example, we have used that interface in the controller, so we can use all the methods which are defined into the interface.



## Benefits of Repository pattern:

1. **Code Reusability and Maintenance:** By using repositories, you can write your database queries or other data access logic once and reuse them throughout your application. This means you don't need to repeat the same query in different places, which can make your code easier to manage and maintain.
2. **Decoupling Model from Controllers:** The repository pattern helps to separate, or decouple, the data access logic from your controllers. This separation helps to make the code more readable and organized.
3. **Abstraction of Data Access:** Repositories provide an abstraction of the data layer. This means that the rest of your application doesn't need to know about the underlying data source or database structure.
4. **Easier Testing:** Because repositories create a level of abstraction away from the database, they can make your code easier to test. You can create mock repositories for your tests instead of having to use the actual database.
5. **Switching Databases:** If in future, you decide to switch your database from MySQL to PostgreSQL or MongoDB, etc., you just need to create a new repository to handle the data operations related to that particular database. You wouldn't need to change your controller or business logic, which can save you a lot of time and effort.

## Factory Pattern:

The Factory Pattern is a design pattern in programming that allows you to create objects without specifying the exact class of object that will be created. This is done by creating a "factory" method that returns instances of different classes.

## Service Pattern:

In Laravel, the Service Pattern isn't a distinct pattern in the same way as the Repository or Factory patterns are. Rather, it's a term that's often used to refer to a style of organizing your code.

When we talk about a "service" in Laravel, we're typically referring to a class that contains some kind of logic that doesn't fit neatly into a controller or model.

Let's take an example: suppose you have an e-commerce application, and you need to implement a feature for processing payments. You could put that logic directly in your controller or model, but it would quickly become messy, especially as you add more features (like handling discounts, taxes, shipping, etc.).

Instead, you might create a `PaymentService` class. This class would contain all the logic related to processing payments, and your controller would use this service to handle payment requests.

```

1 usage new *
class PaymentService
{
    1 usage new *
    public function processPayment($amount, $creditCardInfo)
    {
        // Logic to process the payment...

        return $result;
    }
}

no usages new *
class OrderController extends Controller
{
    2 usages
    protected $paymentService;

    no usages new *
    public function __construct(PaymentService $paymentService)
    {
        $this->paymentService = $paymentService;
    }

    new *
    public function store(Request $request)
    {
        // ...

        $result = $this->paymentService->processPayment($amount, $creditCardInfo);

        // ...
    }
}

```

## Difference of Repository and Service pattern:

The basic difference between the service pattern and the repository pattern lies in their responsibilities and how they are used in the structure of an application.

1. **Repository Pattern:** This pattern is primarily concerned with data access. The repository acts as a bridge between your application and your data source (which is usually a database). It provides a centralized way to read and write data, abstracting away the details of the data source. The benefit of this pattern is that you can change your database or data source with minimal impact on the rest of your code, because all the data access logic is encapsulated in your repositories.
2. **Service Pattern:** This pattern is more about business logic. Service classes typically contain methods that perform operations related to a specific feature of your application. These methods might need to interact with multiple repositories or other services. For example, you might have a `PaymentService` that uses a `UserRepository` to find the current user, a `PaymentRepository` to create a new payment record, and a third-party service to actually process the payment. The advantage of the service pattern is that it helps to keep your controllers and models lean and focused, by offloading complex logic to service classes.

So, in summary:

- Use the Repository Pattern to centralize and abstract away data access.
- Use the Service Pattern to organize and encapsulate complex business logic.

These patterns can and often are used together in the same application, because they serve different purposes and provide different benefits.

## Api Resource:

We use api resources to return api responses in json format, it can be used in many other ways for example if we want

to return specific fields in the response from the collection, we can do that in resource class etc.

## Difference of oauth 2.0 and jwt:

Sure, let's imagine a scenario in which a user wants to log into a third-party application (like a mobile app) using their Google account. Here's how OAuth 2.0 and JWT would be used in that process:

1. The user tries to log into the mobile app. The app doesn't have its own login system, but it offers the option to log in with a Google account.
2. The app redirects the user to a Google sign-in page. This is the beginning of the OAuth 2.0 flow.
3. The user enters their Google credentials. Google then asks the user if they want to grant the mobile app access to their Google account data.
4. If the user agrees, Google creates an access token (which can be a JWT) and returns it to the mobile app. The JWT contains information about the user and the access that the mobile app has been granted. The JWT is signed by Google, which allows the mobile app to verify that the token is legitimate and hasn't been tampered with.
5. The mobile app can now use this JWT to make requests to Google's API on behalf of the user. The JWT is included in the **`Authorization`** header of each request. Google's servers verify the JWT and, if it's valid, allow the request to proceed.
6. When the JWT expires, the mobile app will need to obtain a new one from Google, usually by refreshing the token using a refresh token that was issued along with the original JWT. This process can often be done behind the scenes, without requiring the user to log in again.

This is a simplified example, but it shows how OAuth 2.0 (the process for obtaining access) and JWT (the format of the access token) can be used together to allow a third-party app to access a user's data from a service like Google.

# Difference of authentication and authorization:

Authentication and authorization are two fundamental security concepts that are often used in tandem, but they serve different purposes:

1. **Authentication:** This is the process of verifying who a user is. When a user logs into a system by providing a username and password, the system attempts to authenticate the user by validating that the username and password are correct. If they are correct, the system confirms that the user is who they claim to be. Other forms of authentication can involve biometrics (like fingerprints or facial recognition), tokens, or two-factor authentication.
2. **Authorization:** This is the process of granting or denying a user's access to specific resources or actions within a system once they have been authenticated. Authorization rules can be based on user roles, access levels, or other attributes. For instance, in a blogging system, authenticated users might be authorized to create a new post or edit their own posts, but only users with an "admin" role might be authorized to delete any post.

In summary, while authentication is about verifying the identity of a user, authorization is about deciding what an authenticated user is allowed to do. They are often used together in the phrase "authn & authz", which stands for "authentication and authorization".

# Types of errors in php:

## 1. Parse Errors (Syntax Errors):

- These occur when there's a syntax mistake in the code. They prevent the script from being interpreted correctly.
- Example: Missing semicolons, unmatched braces.
- ``E_PARSE``

## 2. Fatal Errors:

- These are critical errors that halt the execution of the script immediately.
- Example: Calling a non-existent function, instantiating an undefined class.
- ``E_ERROR``

## 3. Warning Errors:

- These errors occur when something goes wrong but does not halt the execution of the script. The script continues to run.
- Example: Including a file that doesn't exist, using incorrect parameters in a function.
- ``E_WARNING``


# Types of arrays:

- Indexed arrays

```
php Copy code  
  
$fruits = ["Apple", "Banana", "Cherry"];  
echo $fruits[0]; // Outputs: Apple
```

- Associative arrays


php

 Copy code

```
$ages = ["Peter" => 35, "Ben" => 37, "Joe" => 43];  
echo $ages["Peter"]; // Outputs: 35
```

- ## Multidimensional Arrays

php

 Copy code

```
$cars = [  
    ["Volvo", 22, 18],  
    ["BMW", 15, 13],  
    ["Saab", 5, 2],  
    ["Land Rover", 17, 15]  
];  
echo $cars[0][0]; // Outputs: Volvo  
echo $cars[2][1]; // Outputs: 5
```

# Array Functions:

## Array Functions

PHP provides a wide array of functions to manipulate arrays, such as:

- ``array_merge()``: Merges one or more arrays.
- ``array_diff()``: Computes the difference between arrays.
- ``array_keys()``: Returns all the keys of an array.
- ``array_values()``: Returns all the values of an array.
- ``array_push()``: Pushes one or more elements onto the end of an array.
- ``array_pop()``: Pops the last element off an array.
- ``array_shift()``: Shifts an element off the beginning of an array.
- ``array_unshift()``: Prepends one or more elements to the beginning of an array.



# Difference of where and when:

## Where:

This is used to add a direct condition to a query. It applies the condition unconditionally and is used in typical query building. For example:

```
$users = User::where('status', 'active')->get();
```

## When:

This is used to conditionally add a query clause. It allows you to apply a condition only if a certain condition is true, making the code cleaner when dealing with optional query parameters. For example:

```
$status = 'active';  
$users = User::when($status, function ($query, $status) {  
    return $query->where('status', $status);  
})->get();
```

## Composer:

### Composer update:

- Reads `composer.json` and updates packages to the latest allowed versions.
- Updates `composer.lock` with new dependency versions.
- Can potentially introduce breaking changes if dependencies update significantly.
- You can update a specific package:

```
sh
```

[Copy](#) [Edit](#)

```
composer update vendor/package-name
```

## Use Case



- When you want to update dependencies to their latest compatible versions.
- After modifying `composer.json` (e.g., adding a new package manually).

## Composer install:

- Reads `composer.lock` and installs the exact versions listed.
- Ensures consistency across different environments.
- **Does not** update `composer.json` or `composer.lock`.

### Example Usage

sh

 Copy  Edit

```
composer install
```

## Use Case



- When cloning a project (`git clone`).
- During deployment to ensure package versions remain the same.

## Composer dump-autoload:

- Rebuilds the `vendor/autoload.php` file.
- Optimizes class loading by regenerating class maps.
- **Does not** install, update, or remove dependencies.

Optimized autoloading for better performance:

sh

 Copy  Edit

```
composer dump-autoload -o
```

## Use Case

- After adding new classes or namespaces manually.
- When experiencing issues with class loading (e.g., "Class not found" errors).

# Bind and singleton:

## Bind:

- Creates a **new instance** every time it is resolved.
- Suitable for services that need fresh instances on each request.

### Example

```
php                                                                    Copy Edit

$this->app->bind(App\Services\PaymentGateway::class, function ($app) {
    return new \App\Services\StripeGateway();
});
```

## Singleton:

- Returns **the same instance** every time it is resolved.
- Useful for shared services like database connections, caching, or loggers.

### Example

```
php                                                                    Copy Edit

$this->app->singleton(App\Services\PaymentGateway::class, function ($app) {
    return new \App\Services\StripeGateway();
});
```