Eloquent ORM (Models & Relationships)

## 1. Basic Concepts

### Model Example

php artisan make:model Post

Eloquent Model Example

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

protected $fillable = ['title', 'body']; // allows mass assignment

}

2. CRUD with Eloquent

Create

Post::create([

'title' => 'My First Post',

'body' => 'This is the content.',

]);

Read

$posts = Post::all(); // get all posts

$post = Post::find(1); // find by primary key

Update

$post = Post::find(1);

$post->title = 'Updated Title';

$post->save();

Delete

Post::destroy(1); // delete by ID

3. Relationships

➤ One to One

// User hasOne Profile

public function profile()

{

return $this->hasOne(Profile::class);

}

// Profile belongsTo User

public function user()

{

return $this->belongsTo(User::class);

}

➤ One to Many

// Post hasMany Comment

public function comments()

{

return $this->hasMany(Comment::class);

}

// Comment belongsTo Post

public function post()

{

return $this->belongsTo(Post::class);

}

➤ Many to Many

// User belongsToMany Role

public function roles()

{

return $this->belongsToMany(Role::class);

}

🔹 4. Relationship Query Examples

// Get all comments of a post

$post = Post::find(1);

$comments = $post->comments;

// Get the post of a comment

$comment = Comment::find(1);

$post = $comment->post;

// Create a comment for a post

$post->comments()->create([

'content' => 'Nice post!',

]);

// Attach role to a user

$user->roles()->attach($roleId);

🔹 5. Eager Loading (for performance)

// Load all posts with their comments

$posts = Post::with('comments')->get();

**2. API development with Laravel Sanctum or Passport**

**🔐 Laravel Sanctum – Recommended for most APIs**

composer require laravel/sanctum

php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"

php artisan migrate

### ✅ 2. Configure Middleware

In app/Http/Kernel.php, add to api middleware group:

\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,

✅ 3. Add HasApiTokens to User model

use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable

{

use HasApiTokens, Notifiable;

}

✅ 4. Token-Based Authentication

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\AuthController;

Route::post('/register', [AuthController::class, 'register']);

Route::post('/login', [AuthController::class, 'login']);

🔹 AuthController Example

use App\Models\User;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Hash;

class AuthController extends Controller

{

public function register(Request $request) {

$user = User::create([

'name' => $request->name,

'email'=> $request->email,

'password' => bcrypt($request->password)

]);

$token = $user->createToken('API Token')->plainTextToken;

return response()->json(['token' => $token], 200);

}

public function login(Request $request) {

$user = User::where('email', $request->email)->first();

if (!$user || !Hash::check($request->password, $user->password)) {

return response()->json(['message' => 'Invalid credentials'], 401);

}

$token = $user->createToken('API Token')->plainTextToken;

return response()->json(['token' => $token], 200);

}

}

✅ 5. Protect Routes with auth:sanctum

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {

return $request->user();

});

🔐 Laravel Passport – Use for Oauth2

✅ 1. Install Passport

composer require laravel/passport

php artisan migrate

php artisan passport:install

✅ 2. Configure Model

use Laravel\Passport\HasApiTokens;

class User extends Authenticatable

{

use HasApiTokens, Notifiable;

}

✅ 3. Configure AuthServiceProvider

✅ 4. In config/auth.php

'guards' => [

'api' => [

'driver' => 'passport',

'provider' => 'users',

],

],

### ✅ 5. Issue Tokens

Use /oauth/token with:

* client\_id
* client\_secret
* grant\_type=password
* username and password

$user = User::find(1);

$token = $user->createToken('MyToken')->accessToken;

**3.Laravel Queues**

### ✅ Use Cases

* Sending emails
* Image/video processing
* Notifications
* Import/export jobs
* API calls to external services

🔧 1. Configuration

QUEUE\_CONNECTION=database

Laravel supports:

* sync (default, processes instantly)
* database
* redis
* sqs, beanstalkd, etc.

📦 2. Set Up Database Queue (if using database)

php artisan queue:table

php artisan migrate

Creates a jobs table to store queued jobs.

🧱 3. Create a Job

php artisan make:job SendWelcomeEmail

This generates app/Jobs/SendWelcomeEmail.php.

Example Job Code

use Illuminate\Bus\Queueable;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Foundation\Bus\Dispatchable;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Queue\SerializesModels;

use App\Models\User;

use Mail;

class SendWelcomeEmail implements ShouldQueue

{

use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

public $user;

public function \_\_construct(User $user)

{

$this->user = $user;

}

public function handle()

{

Mail::to($this->user->email)->send(new \App\Mail\WelcomeMail($this->user));

}

}

🚀 4. Dispatch a Job

use App\Jobs\SendWelcomeEmail;

$user = User::find(1);

SendWelcomeEmail::dispatch($user);

You can also delay it:

SendWelcomeEmail::dispatch($user)->delay(now()->addMinutes(10));

⚙️ 5. Process the Queue

Run this in your terminal:

php artisan queue:work

🛑 6. Retry Failed Jobs

php artisan queue:failed-table

php artisan migrate

php artisan queue:retry all

php artisan queue:failed

### Horizon (for Redis)

If you’re using Redis, Laravel Horizon gives you a beautiful dashboard.

composer require laravel/horizon

php artisan horizon:install

php artisan migrate

php artisan horizon

**4.Events & Listeners**

Laravel’s Events & Listeners system is a clean and powerful way to decouple logic in your application.

## 🎯 Why Use Events & Listeners?

* Separation of concerns: trigger actions without hardcoding them
* Make code modular and reusable
* Execute actions like sending emails, logging, updating stats, etc.

## 🧱 1. Structure

* Event: something that happens (e.g., UserRegistered)
* Listener: reacts to that event (e.g., SendWelcomeEmail)

🔧 2. Create Event & Listener

Automatically with Artisan:

php artisan make:event UserRegistered

php artisan make:listener SendWelcomeEmail –event=UserRegistered

✍️ 3. Define the Event

// app/Events/UserRegistered.php

use App\Models\User;

class UserRegistered

{

use Dispatchable, InteractsWithSockets, SerializesModels;

public $user;

public function \_\_construct(User $user)

{

$this->user = $user;

}

}

✍️ 4. Define the Listener

// app/Listeners/SendWelcomeEmail.php

use App\Events\UserRegistered;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Support\Facades\Mail;

use App\Mail\WelcomeMail;

class SendWelcomeEmail implements ShouldQueue

{

public function handle(UserRegistered $event)

{

Mail::to($event->user->email)->send(new WelcomeMail($event->user));

}

}

🗂 5. Register in EventServiceProvider

// app/Providers/EventServiceProvider.php

protected $listen = [

\App\Events\UserRegistered::class => [

\App\Listeners\SendWelcomeEmail::class,

],

];

🚀 6. Fire the Event

use App\Events\UserRegistered;

$user = User::create([...]);

event(new UserRegistered($user));

🎛 7. Optional: Manual Listener Command

php artisan queue:work

**5.Laravel Scheduler (CRON jobs)**

Laravel's Task Scheduler makes it easy to run periodic tasks using CRON—without writing full crontab entries for each command.

🔧 1. Add Scheduler to Server CRON

Run this in your terminal (replace path accordingly):

\* \* \* \* \* php /path-to-your-project/artisan schedule:run >> /dev/null 2>&1

🧱 2. Define Scheduled Tasks

protected function schedule(Schedule $schedule)

{

$schedule->command('inspire')->hourly();

}

Common Examples:

$schedule->command('emails:send')->daily();

$schedule->call(function () {

DB::table('users')->delete();

})->sundays();

$schedule->exec('node /home/script.js')->everyMinute();

$schedule->job(new AutoBackupJob)->weekly();

## ⏰ 3. Available Time Intervals

| Method | Description |
| --- | --- |
| ->everyMinute() | Every minute |
| ->hourly() | Every hour |
| ->daily() | Once per day |
| ->weekly() | Once per week |
| ->monthly() | Once per month |
| ->cron('\* \* \* \* \*') | Custom CRON expression |

✅ 4. Run Commands or Jobs

$schedule->command('cache:clear')->dailyAt('03:00');

🔹 Inline Closure

$schedule->call(function () {

Log::info('Task ran at ' . now());

})->everyMinute();

🔹 Dispatch Job

use App\Jobs\AutoInvoiceJob;

$schedule->job(new AutoInvoiceJob)->monthlyOn(1, '02:00');

📜 5. Schedule Custom Commands

php artisan make:command SendNewsletter

In app/Console/Commands/SendNewsletter.php, implement the handle() method, then register it in the scheduler:

$schedule->command('newsletter:send')->mondays()->at('08:00');

🔐 6. Task Constraints

$schedule->command('report:generate')->withoutOverlapping();

Run only on production:

$schedule->command('backup:run')->environments(['production']);

🧪 7. Test the Scheduler Manually

php artisan schedule:run

**6.Caching and Performance Optimization**

## ⚡ 1. **Types of Caching in Laravel**

| Type | Use Case | Command/API |
| --- | --- | --- |
| Config Cache | Cache config files | php artisan config:cache |
| Route Cache | Cache routes | php artisan route:cache |
| View Cache | Cache Blade templates | php artisan view:cache |
| Application Cache | Store custom data like DB results | Cache::put(), Cache::remember() |
| Query Cache | Cache expensive DB queries | Use remember() with Eloquent or DB |
| Response Cache | Cache full HTTP responses | Use package like spatie/laravel-responsecache |

🔧 2. Set Cache Driver

CACHE\_DRIVER=redis # or file, database, memcached

Then, configure in config/cache.php.

🧠 3. Application Cache (Code Examples)

Cache a value:

Cache::put('key', 'value', now()->addMinutes(10));

Get with default:

$value = Cache::get('key', 'default');

Remember pattern (auto-fetch if not cached):

$data = Cache::remember('users.all', 600, function () {

return DB::table('users')->get();

});

🧮 4. Eloquent Query Caching

$users = Cache::remember('users.active', 60, function () {

return User::where('status', 'active')->get();

});

🖼️ 5. View Caching

php artisan view:cache

Clears with:

php artisan view:clear

🗺️ 6. Route & Config Caching

php artisan route:cache

php artisan config:cache

php artisan config:clear

Always run after changing routes/web.php or .env.

🧪 7. Database Query Optimization

Use eager loading to avoid N+1:

$posts = Post::with('comments')->get();

Index your columns (via migrations):

$table->index('email');

Avoid heavy joins; use efficient queries.

## 🚀 8. **Performance Tips**

### Assets & Frontend:

* Use Laravel Mix / Vite to minify CSS & JS.
* Lazy-load images, reduce DOM size.

### Jobs & Queues:

* Move heavy tasks (emails, API calls) to queues.
* Use ShouldQueue for listeners.

### DB Optimizations:

* Optimize slow queries (check with DB::listen() or Laravel Telescope)
* Use pagination (paginate()), not all() for large data.

### Use OPCache (PHP):

* Enable PHP OPCache in production.

### Use Redis or Memcached:

* Configure as cache and session driver for high performance.

## 🧰 9. Tools for Monitoring

* 🔍 Laravel Debugbar: shows query performance, memory usage, etc.
* 🧭 Laravel Telescope: full app monitoring (requests, jobs, DB, etc.)
* 📈 Clockwork: Chrome extension + package for debugging.

**7.How to use middleware to protect routes?**

In Laravel, middleware acts as a filter between a request and a response. You can use it to protect routes, check authentication, roles, permissions, or even throttle API usage.

## 🔐 Common Use: Protect Routes with auth Middleware

### ✅ Step 1: Apply auth Middleware

In routes/web.php or routes/api.php:

Route::middleware('auth')->group(function () {

Route::get('/dashboard', [DashboardController::class, 'index']);

});

Or for a single route:

Route::get('/profile', [UserController::class, 'show'])->middleware('auth');

🛠 Step 2: Ensure Middleware is Registered

protected $routeMiddleware = [

'auth' => \App\Http\Middleware\Authenticate::class,

'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,

'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,

'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,

// Add your custom middleware here

];

👮 Custom Middleware

Step 1: Create Middleware

php artisan make:middleware EnsureUserIsAdmin

Step 2: Define Logic

// app/Http/Middleware/EnsureUserIsAdmin.php

public function handle($request, Closure $next)

{

if (auth()->check() && auth()->user()->is\_admin) {

return $next($request);

}

return response()->json(['error' => 'Unauthorized'], 403);

}

Step 3: Register Middleware

In app/Http/Kernel.php:

'admin' => \App\Http\Middleware\EnsureUserIsAdmin::class,

Step 4: Use It

Route::middleware(['auth', 'admin'])->group(function () {

Route::get('/admin/dashboard', [AdminController::class, 'index']);

});

### 1. **Explain Laravel’s service container and how dependency injection works.**

* Service Container is a powerful tool in Laravel used for managing class dependencies and performing dependency injection.
* It acts as a registry that knows how to instantiate and inject the objects your application needs.
* When a class requires dependencies (via constructor or method), Laravel resolves them automatically by inspecting type hints and providing the required objects.
* This enables loose coupling and easier testing.

Example:

class UserController {

protected $userService;

// Laravel automatically injects UserService here

public function \_\_construct(UserService $userService) {

$this->userService = $userService;

}

}

### 2. **What is a service provider in Laravel and how do you use it?**

* A Service Provider is a central place to register bindings and bootstrapping code within the Laravel service container.
* They are the entry point for all Laravel application bootstrapping.
* Laravel’s core and your custom services are bootstrapped through service providers.
* You register them in config/app.php (most often auto-registered via package discovery).

Example use cases:

* Binding interfaces to implementations
* Registering event listeners
* Configuring packages
* Adding middleware, commands, routes

Example:

class AppServiceProvider extends ServiceProvider

{

public function register()

{

// Binding interface to implementation

$this->app->bind(

UserRepositoryInterface::class,

EloquentUserRepository::class

);

}

public function boot()

{

// Code to run after all services are registered

}

}

### 3. **How does Laravel handle facades under the hood?**

* Facades provide a static interface to classes available in the service container.
* When you call a facade method like Cache::get(), Laravel routes that call to an instance resolved from the container.
* Behind the scenes, facades extend Illuminate\Support\Facades\Facade.
* The key part is the getFacadeAccessor() method which returns the service container binding name.
* The facade uses PHP's \_\_callStatic magic method to forward calls to the underlying instance.

Simplified flow:

Cache::get('key');

↓

Facade::\_\_callStatic('get', ['key']);

↓

Resolve 'cache' service from container

↓

Call `get('key')` on resolved cache instance

### 4. **How does Laravel's request lifecycle work from HTTP request to response?**

1. Request received by the webserver (Apache/Nginx).
2. Passed to public/index.php (Laravel’s front controller).
3. Laravel creates a Request object (Illuminate\Http\Request).
4. Request goes through middleware stack (global + route middleware).
5. Request is dispatched to the router (Illuminate\Routing\Router) which matches it to a route.
6. Route calls the associated controller method or closure.
7. Controller processes request, uses services, models, etc.
8. Controller returns a response (view, JSON, redirect).
9. Response goes back through middleware (for post-processing).
10. Response is sent back to the browser.

### 5. **Difference between service container and service provider?**

| Feature | Service Container | Service Provider |
| --- | --- | --- |
| What it is | A container for managing class dependencies and resolving them | A class responsible for registering bindings and bootstrapping |
| Role | Manages objects and dependencies at runtime | Registers services, bindings, and initial setup |
| Usage | Resolves dependencies for classes (via injection) | Registers bindings, event listeners, routes, etc. |
| Where used | Throughout the app wherever dependencies are injected | Loaded during app bootstrapping (config/app.php) |
| Example | $app->make(UserRepository::class) | AppServiceProvider::register() binds interface to implementation |

### 6. **How do you bind interfaces to implementations in Laravel?**

You bind in a service provider inside the register() method using the container’s bind or singleton method.

Example:

$this->app->bind(

\App\Repositories\UserRepositoryInterface::class,

\App\Repositories\EloquentUserRepository::class

);

When the container sees a constructor type hint for UserRepositoryInterface, it injects an instance of EloquentUserRepository.

* Use bind() to create new instances on each request or singleton() to share the same instance.

### 1. **What are the differences between** hasOneThrough **and hasManyThrough?**

* Both are "through" relationships used to access a related model through an intermediate model.

| Aspect | hasOneThrough | hasManyThrough |
| --- | --- | --- |
| Relationship type | One-to-one through another model | One-to-many through another model |
| Usage example | Get user's phone through the user’s country | Get user's posts through the user's blog |
| Returns | Single related model instance | Collection of related model instances |
| Method signature | hasOneThrough(FinalModel::class, IntermediateModel::class, ...) | hasManyThrough(FinalModel::class, IntermediateModel::class, ...) |

Example:

// User has one Phone through Country

public function phone()

{

return $this->hasOneThrough(Phone::class, Country::class);

}

// Country has many Posts through User

public function posts()

{

return $this->hasManyThrough(Post::class, User::class);

}

### 2. **How do you eager load nested relationships?**

You eager load nested relationships using dot notation with with().

// Eager load posts with their comments and each comment's author

$posts = Post::with('comments.author')->get();

You can also eager load multiple nested relationships:

$users = User::with(['posts.comments', 'profile'])->get();

### 3. **How would you handle complex query filters in a large Eloquent model?**

* Use query scopes (local or global) to encapsulate reusable filters.
* Use Repository Pattern or Query Builders to separate query logic.
* Use conditional queries with when() to apply filters only when needed.
* For very complex filtering, consider custom query classes or specifications.

Example with query scopes:

class User extends Model

{

public function scopeActive($query)

{

return $query->where('active', true);

}

public function scopeFilterByRole($query, $role)

{

return $query->where('role', $role);

}

}

// Usage:

$users = User::active()->filterByRole('admin')->get();

### 4. **What is the difference between update() and updateOrCreate()?**

| Method | Purpose | Behavior |
| --- | --- | --- |
| update() | Updates existing model(s) based on a query | Returns number of affected rows |
| updateOrCreate() | Tries to find a record matching attributes; if found, updates it; else creates a new record | Returns the model instance (updated or newly created) |

Example:

// Update all active users to role 'admin'

User::where('active', true)->update(['role' => 'admin']);

// Find user by email, update if found, or create if not

$user = User::updateOrCreate(

['email' => 'user@example.com'],

['name' => 'New Name', 'role' => 'admin']

);

### 5. **How do you use accessors, mutators, and attribute casting?**

* Accessors transform model attribute values when retrieved.

public function getFullNameAttribute()

{

return $this->first\_name . ' ' . $this->last\_name;

}

Usage: $user->full\_name

* Mutators modify attribute values before saving to the database.

public function setPasswordAttribute($value)

{

$this->attributes['password'] = bcrypt($value);

}

Attribute Casting defines how attributes are automatically cast to native types.

protected $casts = [

'is\_admin' => 'boolean',

'created\_at' => 'datetime',

'options' => 'array',

];

### 6. **How to optimize Eloquent to avoid N+1 query problem?**

* The N+1 problem occurs when you load a collection and then lazily load a related model for each item, resulting in many queries.
* Use eager loading with with() to load related models in a single query.

Example without eager loading (N+1 problem):

$books = Book::all();

foreach ($books as $book) {

echo $book->author->name; // One query per book for author

}

Optimized with eager loading:

$books = Book::with('author')->get();

foreach ($books as $book) {

echo $book->author->name; // Loaded in 2 queries: one for books, one for authors

}

For nested relations, use nested eager loading:

Order::with('items.product')->get();

Use debugging tools like Laravel Debugbar or clock queries to detect N+1 problems.

### 1. **Compare Laravel Sanctum vs Passport – when to use which?**

| Feature | Laravel Sanctum | Laravel Passport |
| --- | --- | --- |
| Protocol | Uses simple token-based authentication (API tokens + SPA cookie-based auth) | Full OAuth2 server implementation |
| Complexity | Lightweight and easy to set up | More complex, suited for OAuth2 requirements |
| Use cases | Single-page applications (SPAs), mobile apps, simple token APIs | Large-scale APIs requiring OAuth2 flows (authorization code, client credentials) |
| Token types | Personal access tokens, SPA session cookies | Access tokens, refresh tokens, client tokens |
| OAuth2 Compliance | No | Yes |
| Token Scopes | Supports simple token abilities/scopes | Full OAuth2 scopes and refresh tokens |
| Refresh Tokens | No (tokens are long-lived or can be revoked) | Yes |
| Third-party integration | No OAuth2 client support | Supports third-party apps as OAuth2 clients |

### 2. **How does token-based authentication work in Laravel?**

* Client (mobile app, SPA) sends credentials (email/password) to API.
* Laravel verifies credentials, generates a token (via Sanctum/Passport or custom).
* The token is returned to client, stored securely (localStorage, secure cookie).
* Client sends token with each API request (in Authorization: Bearer <token> header).
* Laravel middleware authenticates request by validating token.
* On valid token, Laravel sets the authenticated user (Auth::user()).
* Token may have scopes/abilities defining allowed actions.
* Tokens can be revoked or expired.

### 3. **How do you build role & permission management? Use any package?**

* Common approach: Use spatie/laravel-permission package (widely used, well-maintained).

Features:

* Define roles and assign permissions.
* Assign roles or permissions to users.
* Middleware to check roles/permissions.
* Blade directives for authorization.
* Cache for fast permission checking.

Basic usage:

composer require spatie/laravel-permission

php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"

php artisan migrate

use Spatie\Permission\Models\Role;

use Spatie\Permission\Models\Permission;

$role = Role::create(['name' => 'admin']);

$permission = Permission::create(['name' => 'edit articles']);

$role->givePermissionTo($permission);

$user->assignRole('admin');

Checking in code:

if ($user->can('edit articles')) { ... }

if ($user->hasRole('admin')) { ... }

### 4. **What are policies and gates in Laravel, and when do you use each?**

| Concept | Description | Use Case |
| --- | --- | --- |
| Gate | Closure-based authorization checks | Simple authorization logic not tied to a model |
| Policy | Class-based authorization tied to a model | Complex model-specific authorization logic |

Gate example:

Gate::define('update-post', function ($user, $post) {

return $user->id === $post->user\_id;

});

Policy example:

php artisan make:policy PostPolicy –model=Post

In PostPolicy:

public function update(User $user, Post $post)

{

return $user->id === $post->user\_id;

}

Register policy in AuthServiceProvider:

protected $policies = [

Post::class => PostPolicy::class,

];

Use in controller:

$this->authorize('update', $post);

### 5. **How do you secure APIs against CSRF, XSS, SQL injection in Laravel?**

* CSRF (Cross-Site Request Forgery):
  + Laravel has built-in CSRF protection via VerifyCsrfToken middleware.
  + For APIs, use token-based auth (Sanctum SPA or Passport), which is immune to CSRF since tokens are sent via headers.
  + For form POST requests, include @csrf Blade directive or add token header.
* XSS (Cross-Site Scripting):
  + Escape output using Blade syntax: {{ $variable }} escapes HTML by default.
  + Use e() helper for manual escaping.
  + Use input validation to disallow dangerous HTML or sanitize inputs.
* SQL Injection:
  + Always use Eloquent ORM or Laravel’s Query Builder, which use parameter binding automatically.
  + Avoid raw queries with direct user input; if raw queries are necessary, use bindings:

DB::select('select \* from users where email = ?', [$email]);

Additionally:

* Validate all inputs rigorously.
* Use HTTPS to encrypt data.
* Set HTTP security headers using middleware.

### 1. **Explain the lifecycle of a queued job in Laravel**

1. Job Creation:  
   You create a job class (e.g., php artisan make:job SendEmail) that implements ShouldQueue.
2. Dispatching:  
   The job is dispatched using dispatch() or helper methods. This pushes the job payload into the configured queue backend (database, Redis, etc.).
3. Queue Storage:  
   The job is stored in the queue system (e.g., jobs table, Redis list).
4. Queue Worker:  
   A queue worker process (php artisan queue:work) listens to the queue and fetches jobs when available.
5. Job Execution:  
   The worker unserializes the job class and calls the handle() method.
6. Job Completion:
   * If the job executes successfully, it is removed from the queue.
   * If the job fails (throws an exception), Laravel will retry it based on retry configuration.
7. Failed Job Handling:  
   If the job fails after all retry attempts, it is moved to the failed jobs table (if configured).

2. How do you handle failed jobs and retries?

class SendEmail implements ShouldQueue

{

public $tries = 5; // Number of retry attempts

public $timeout = 120; // Timeout in seconds

}

Configure failed jobs storage:

php artisan queue:failed-table

php artisan migrate

.Laravel automatically moves jobs to the failed\_jobs table after max retries.

* To manually retry failed jobs:

php artisan queue:retry {job\_id}

To delete failed jobs:

php artisan queue:forget {job\_id}

You can listen to failed job events for custom handling:

Queue::failing(function (JobFailed $event) {

// Log or notify admin

});

### 3. **How do you use events and listeners for decoupling logic?**

* Events represent actions or changes in your app (e.g., UserRegistered).
* Listeners respond to those events (e.g., send welcome email).

This decouples core logic from side effects.

Example:

php artisan make:event UserRegistered

php artisan make:listener SendWelcomeEmail –event=UserRegistered

Dispatch event in code:

event(new UserRegistered($user));

protected $listen = [

UserRegistered::class => [

SendWelcomeEmail::class,

],

];

Benefits:

* Single Responsibility: Events trigger listeners without tight coupling.
* Easy to add/remove functionality.
* Listeners can be queued for async processing.

### 4. **What is the difference between** dispatch() **and** dispatchNow()**?**

| Method | Behavior | Use case |
| --- | --- | --- |
| dispatch() | Pushes the job to the queue to be processed asynchronously by a worker | For background/asynchronous processing |
| dispatchNow() | Executes the job immediately in the current process (synchronously) | For immediate execution without queuing (usually testing or urgent) |

SendEmailJob::dispatch($user); // Queued, async

SendEmailJob::dispatchNow($user); // Runs immediately, blocking

### 5. **How do you handle real-time notifications or broadcasting in Laravel?**

* Laravel uses Broadcasting to send real-time events via WebSockets.
* Supported drivers: Pusher, Laravel Echo Server (Node), Redis, etc.

Steps:

1. Create an event that implements ShouldBroadcast:

php artisan make:event MessageSent

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class MessageSent implements ShouldBroadcast

{

public $message;

public function \_\_construct($message)

{

$this->message = $message;

}

public function broadcastOn()

{

return new PrivateChannel('chat');

}

}

Configure broadcasting in config/broadcasting.php.

Set up Laravel Echo (frontend) to listen for events:

Echo.private('chat')

.listen('MessageSent', (e) => {

console.log(e.message);

});

Authorize private channels in routes/channels.php:

Broadcast::channel('chat', function ($user) {

return Auth::check();

});

### **1. What types of testing does Laravel support? (Feature vs Unit)**

* Unit Tests:
  + Test individual classes or methods in isolation.
  + No database or external services.
  + Fast and focused on small code units.
  + Usually placed in tests/Unit.
* Feature Tests:
  + Test larger parts of your app, including HTTP requests, database, middleware, views.
  + Often hit the database (uses testing database).
  + Test app behavior as a whole (e.g., submitting a form, user login).
  + Usually placed in tests/Feature.

Example:

php artisan make:test UserTest --unit

php artisan make:test LoginTest

### **2. How do you mock dependencies in a test?**

* Laravel integrates Mockery for mocking.
* You can mock dependencies (services, repositories) to isolate the unit under test.

public function test\_service\_is\_called()

{

$mock = \Mockery::mock(SomeService::class);

$mock->shouldReceive('performAction')

->once()

->andReturn('mocked result');

$this->app->instance(SomeService::class, $mock);

$response = $this->get('/some-route');

$response->assertStatus(200);

}

You can also mock facades easily:

use Illuminate\Support\Facades\Mail;

Mail::fake(); // Prevents actual emails, lets you assert

Mail::assertSent(SomeMailable::class);

### **3. How do you test queued jobs or events in Laravel?**

* Testing queued jobs:

Use Bus::fake() to fake dispatching jobs and assert they were dispatched.

use Illuminate\Support\Facades\Bus;

public function test\_job\_is\_dispatched()

{

Bus::fake();

// Trigger action that dispatches job

$this->post('/send-email');

Bus::assertDispatched(SendEmailJob::class);

}

Testing events:

use Illuminate\Support\Facades\Event;

public function test\_event\_fired()

{

Event::fake();

$this->post('/register');

Event::assertDispatched(UserRegistered::class);

}

### **4. How do you use Laravel Telescope and Debugbar?**

* Laravel Telescope:
  + A debug assistant for Laravel apps.
  + Records requests, exceptions, database queries, jobs, events, cache, mail, notifications, etc.
  + Install via:

composer require laravel/telescope --dev

php artisan telescope:install

php artisan migrate

php artisan serve

Access via /telescope URL.

Great for development to monitor detailed app internals and performance.

Laravel Debugbar:

* Adds a developer toolbar in the browser.
* Shows debug info about queries, routes, views, timeline, session, etc.

Install:

composer require barryvdh/laravel-debugbar –dev

## **Using Laravel with Docker**

### Why Docker for Laravel?

* Provides consistent development environment regardless of OS.
* Easy to replicate production environment.
* Simplifies dependency management (PHP, DB, queue workers, cache).
* Easier onboarding for new developers.

### Basic Laravel Docker Setup

1. Dockerfile: Defines the PHP environment with extensions needed for Laravel.

FROM php:8.1-fpm

# Install dependencies

RUN apt-get update && apt-get install -y \

libonig-dev \

libzip-dev \

zip \

unzip \

git \

curl

# Install PHP extensions

RUN docker-php-ext-install pdo\_mysql mbstring zip exif pcntl bcmath

# Install Composer

COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

WORKDIR /var/www/html

COPY . .

RUN composer install --no-dev –optimize-autoloader

docker-compose.yml: To orchestrate Laravel app, MySQL, Redis, etc.

version: '3.8'

services:

app:

build:

context: .

dockerfile: Dockerfile

volumes:

- .:/var/www/html

ports:

- 8000:8000

depends\_on:

- db

- redis

environment:

- DB\_HOST=db

- REDIS\_HOST=redis

db:

image: mysql:8.0

environment:

MYSQL\_ROOT\_PASSWORD: rootpass

MYSQL\_DATABASE: laravel

MYSQL\_USER: laraveluser

MYSQL\_PASSWORD: laravelpass

volumes:

- dbdata:/var/lib/mysql

redis:

image: redis:alpine

volumes:

dbdata:

Run containers:

docker-compose up -d

Inside container, run migrations:

docker exec -it <app\_container\_name> php artisan migrate

## Using Laravel with CI/CD Pipelines

### What is CI/CD?

* CI (Continuous Integration): Automated build, test, and validation on every code push.
* CD (Continuous Deployment/Delivery): Automated deployment of code to staging/production after successful tests.

### Typical Laravel CI/CD Workflow

1. Code push triggers pipeline (GitHub Actions, GitLab CI, Jenkins, CircleCI).
2. CI Pipeline steps:
   * Checkout code.
   * Install PHP dependencies (composer install).
   * Run code quality tools (PHPStan, PHP-CS-Fixer).
   * Run unit and feature tests (php artisan test or vendor/bin/phpunit).
   * Run static analysis and linting.
   * Build assets (npm install && npm run prod if applicable).
   * Cache config/routes/views.
   * Optionally build Docker image.
3. CD Pipeline steps:
   * Deploy code to server (via SSH, FTP, or container registry).
   * Run database migrations (php artisan migrate).
   * Clear and cache config/routes/views.
   * Restart queue workers if any.
   * Run any post-deployment scripts.

Example: GitHub Actions workflow for Laravel

name: Laravel CI

on: [push, pull\_request]

jobs:

build:

runs-on: ubuntu-latest

services:

mysql:

image: mysql:8.0

env:

MYSQL\_ROOT\_PASSWORD: root

MYSQL\_DATABASE: testdb

ports:

- 3306:3306

options: >-

--health-cmd="mysqladmin ping"

--health-interval=10s

--health-timeout=5s

--health-retries=3

steps:

- uses: actions/checkout@v2

- name: Setup PHP

uses: shivammathur/setup-php@v2

with:

php-version: 8.1

extensions: mbstring, pdo\_mysql, zip

- name: Install dependencies

run: composer install --prefer-dist --no-progress --no-suggest --no-interaction

- name: Create .env

run: cp .env.example .env

- name: Generate app key

run: php artisan key:generate

- name: Run migrations

run: php artisan migrate --force

- name: Run tests

run: php artisan test –verbose

### 1. **Laravel Horizon**

* Purpose:  
  A beautiful dashboard and code-driven configuration for managing Laravel queues (Redis-backed queues specifically).
* Experience Highlights:
  + Set up real-time queue monitoring and metrics (job throughput, failures, runtime).
  + Used Horizon’s dashboards to quickly identify failing jobs and retries.
  + Configured multiple queue supervisors for different job types (emails, notifications, reports).
  + Managed job prioritization and balancing worker processes.
  + Set up notifications for failed jobs via Slack or email.
  + Used Horizon’s retry and job release features to handle transient errors.
  + Monitored queue performance and scaled workers accordingly.

### 2. **Laravel Nova**

* Purpose:  
  A premium admin panel builder for Laravel applications with resource management, filters, lenses, metrics, and authorization built-in.
* Experience Highlights:
  + Quickly scaffolded admin interfaces for managing users, products, orders, etc.
  + Customized resource fields, filters, and actions for complex admin workflows.
  + Integrated Nova with policies and gates to enforce permissions.
  + Created custom tools and cards to extend Nova dashboards.
  + Used Nova’s resource metrics and trends for business insights.
  + Deployed Nova as a back-office for clients to manage data without building custom UI.
  + Handled file uploads and relationships elegantly using Nova fields.

### 3. **Laravel Cashier**

* Purpose:  
  Subscription billing management with Stripe (and Paddle in newer versions), simplifying recurring payments, coupons, invoices, and more.
* Experience Highlights:
  + Implemented subscription plans with trial periods and billing cycles.
  + Managed customers’ payment methods and invoices.
  + Handled subscription upgrades, downgrades, cancellations, and renewals.
  + Applied coupon codes and discounts seamlessly.
  + Used webhooks to sync subscription events and update user status.
  + Generated invoices and receipts for users.
  + Integrated Cashier with multi-tenant SaaS applications.

### 1. **How would you architect a scalable API for a multi-tenant SaaS app?**

* Tenant Isolation Strategy:
  + Use single database with tenant\_id column for most apps (easier to maintain, cost-effective).
  + Or separate database per tenant for high data isolation/security or huge datasets.
  + Use packages like [tenancy/tenancy](https://tenancy.dev/) or [spatie/laravel-multitenancy] to help manage tenants.
* API Design:
  + Use API versioning to handle backward compatibility.
  + Authenticate tenants via API tokens or OAuth.
  + Use scoped queries that filter data by tenant ID automatically.
  + Implement rate limiting per tenant to prevent abuse.
  + Use resource-based authorization (Policies/Gates) for tenant-level access control.
* Performance & Scalability:
  + Cache tenant-specific config, permissions, and API responses.
  + Queue long-running jobs per tenant with priority queues.
  + Use horizontal scaling with load balancers.
  + Monitor per-tenant usage and health.
* Security:
  + Ensure tenant data cannot leak by scoping queries.
  + Sanitize and validate all input per tenant context.
  + Use encryption for sensitive tenant data.

### 2. **If a job fails intermittently due to API timeout, how do you solve it?**

* Increase timeout limits in HTTP client configuration.
* Use retry logic with exponential backoff to avoid overwhelming the API.
* Implement circuit breaker pattern to pause retries if failure threshold is met.
* Catch exceptions and fail the job gracefully, allowing Laravel to retry later.
* Log detailed error messages and timestamps.
* Consider splitting the job into smaller chunks if processing too much data at once.
* Monitor external API status or switch to alternative endpoints if available.

### 3. **A customer complains about slow API — how would you debug and fix it?**

* Collect logs and metrics:
  + Use Laravel Telescope, Debugbar, or APM tools like New Relic.
  + Check database query count and execution time for the request.
  + Profile application code to find bottlenecks.
* Common issues to check:
  + N+1 query problems — use eager loading.
  + Large or unindexed database queries.
  + External API calls blocking request.
  + Slow file I/O or cache misses.
  + High server CPU/memory usage.
* Fixes:
  + Optimize or add indexes to DB.
  + Cache frequent queries or responses.
  + Offload heavy processing to queues.
  + Use Redis or Memcached for caching.
  + Use pagination or limit returned data.
  + Scale infrastructure horizontally if needed.

### 4. **How would you migrate a legacy Laravel 5 app to Laravel 11 safely?**

* Preparation:
  + Backup the full project and database.
  + Set up a parallel environment for migration.
  + Review Laravel 5 to Laravel 11 upgrade guides for breaking changes.
* Migration steps:
  + Upgrade PHP version to meet Laravel 11 requirements.
  + Incrementally upgrade Laravel versions (e.g., 5.x → 6 → 8 → 10 → 11), testing at each step.
  + Update composer dependencies accordingly.
  + Refactor deprecated features (middleware, routes, helper functions).
  + Replace legacy packages with maintained ones.
  + Run all tests and fix failures.
  + Migrate config files and environment variables.
  + Update database migrations and seeders as needed.
  + Perform manual testing and QA.
  + Deploy the migrated app parallelly and do a staged rollout.

Laravel Forge is a server management and deployment service designed specifically for Laravel applications. It simplifies provisioning, configuring, and deploying PHP servers without needing to manage everything manually.

### What is Laravel Forge?

* A SaaS platform by the Laravel creator Taylor Otwell.
* Helps developers provision servers on popular cloud providers like DigitalOcean, Linode, AWS, Vultr, and more.
* Automates setting up the server environment with PHP, Nginx, MySQL/MariaDB, Redis, SSL certificates, and other services needed to run Laravel apps.
* Provides one-click deployment of your Laravel projects from GitHub, GitLab, or Bitbucket.
* Supports managing multiple sites, databases, SSL certificates (Let's Encrypt), queues, and workers.
* Simplifies zero-downtime deployments.
* Includes tools for server monitoring, scheduled tasks (cron jobs), and team collaboration.

### Why use Forge?

* Saves time on server setup and configuration.
* No need to manually SSH and install dependencies.
* Makes Laravel app deployment repeatable, consistent, and less error-prone.
* Enables quick scaling or migration by managing multiple servers.
* Great for solo developers and teams to manage production or staging environments efficiently.