01_Laravel_Advanced_Routing.md

Implement Dynamic Route Model Binding with Custom Keys and Route Groups with Middleware in Laravel

Objective

Learn how to:

1. Use **custom keys** in **Route Model Binding** (e.g., `slug` instead of `id`).
2. Group routes with **middleware** for authentication, rate limiting, etc.

---

1. Dynamic Route Model Binding with Custom Keys

By default, Laravel resolves Eloquent models using `id`. To use a different column (e.g., `slug`), modify:

Option 1: Explicit Binding in `RouteServiceProvider`

```
// app/Providers/RouteServiceProvider.php
public function boot()
{
    parent::boot();

    // Bind 'post' parameter to Post model using 'slug' instead of 'id'
    Route::model('post', \App\Models\Post::class);
    Route::bind('post', function ($slug) {
        return \App\Models\Post::where('slug', $slug)->firstOrFail();
    });
}
```

PROF

Option 2: Override `getRouteKeyName()` in the Model

```
// app/Models/Post.php
class Post extends Model
{
    public function getRouteKeyName()
    {
        return 'slug'; // Use 'slug' instead of 'id' for URLs
    }
}
```

Usage in Routes

```
// routes/web.php
Route::get('/posts/{post}', function (Post $post) {
    return view('posts.show', compact('post'));
});
```

- Now `/posts/my-post-slug` will fetch the post where `slug = 'my-post-slug'`.

---

2. Route Groups with Middleware

Group routes to apply **middleware** (e.g., `auth`, `throttle`, custom middleware).

Example: Admin Routes with Auth & Custom Middleware

```
// routes/web.php
Route::middleware(['auth', 'admin'])->prefix('admin')->group(function ()
{
    Route::get('/dashboard', [AdminController::class, 'dashboard']);
    Route::get('/users', [AdminController::class, 'users']);
});
```

- **`auth`**: Ensures the user is logged in.
- **`admin`**: A custom middleware (e.g., checks if `user->is_admin = true`).
- **`prefix('admin')`**: Prepends `/admin` to all routes in the group.

Example: API Routes with Rate Limiting

```
// routes/api.php
Route::middleware('throttle:60,1')->group(function () {
    Route::get('/posts', [PostController::class, 'index']);
    Route::get('/posts/{post}', [PostController::class, 'show']);
});
```

PROF

- **`throttle:60,1`**: Limits to 60 requests per minute per IP.

Final Implementation

1. Custom Route Binding + Middleware Group

```
// routes/web.php
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'show']);
    // Uses 'slug' instead of 'id' due to RouteServiceProvider or Model
setup
    Route::get('/posts/{post}', [PostController::class, 'show']);
});
```

2. Testing

- Visit `/posts/my-post-slug` → Should fetch the correct post.
- Visit `/admin/dashboard` → Should redirect if not logged in or not an admin.

Key Takeaways

✅ **Custom Route Binding**: Fetch models using `slug`, `username`, etc., instead of `id`.
✅ **Route Groups**: Apply middleware, prefixes, and namespaces to multiple routes efficiently.
✅ **Middleware**: Restrict access (e.g., `auth`, `admin`, `throttle`).

Next Step: Try implementing **Laravel Policies & Gates** for fine-grained authorization! 🚀

02_Laravel_Service_Providers.md

Create a Custom Service Provider to Register a Singleton Service and Defer Its Loading

Objective

Learn how to:

1. **Create a custom Service Provider** in Laravel.
2. **Register a singleton service** (only one instance throughout the request lifecycle).
3. **Defer loading** to improve performance.

PROF

---

1. Create a Custom Service

First, define a service class (e.g., `PaymentGateway`).

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

class PaymentGateway
{
    public function charge(float $amount): bool
    {
        // Simulate payment processing
        return true;
    }
}
```

2. Generate a Custom Service Provider

Run:

```
php artisan make:provider PaymentServiceProvider
```

This creates:

```
// app/Providers/PaymentServiceProvider.php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Services\PaymentGateway;

class PaymentServiceProvider extends ServiceProvider
{
    public function register()
    {
        // Register as a singleton (single instance)
        $this->app->singleton(PaymentGateway::class, function ($app) {
            return new PaymentGateway();
        });
    }

    // Optional: Defer loading until needed
    public function provides()
    {
        return [PaymentGateway::class];
    }
}
```

PROF

- `singleton()`: Ensures only one instance is created per request.
- `provides()`: Defines which services are deferred.

3. Register the Service Provider

Add to `config/app.php`:

```
'providers' => [
    // ...
    App\Providers\PaymentServiceProvider::class,
],
```

---

4. Defer Loading (Optional)

To **delay loading** until the service is actually used:

```
// app/Providers/PaymentServiceProvider.php
class PaymentServiceProvider extends ServiceProvider
{
    protected $defer = true; // Defer loading

    public function provides()
    {
        return [PaymentGateway::class];
    }
}
```

- `$defer = true`: Only loads when explicitly requested.

---

5. Use the Singleton Service

Now, inject `PaymentGateway` anywhere (controllers, jobs, etc.):

Method 1: Dependency Injection

```
// app/Http/Controllers/PaymentController.php
use App\Services\PaymentGateway;

public function pay(PaymentGateway $paymentGateway)
{
    $success = $paymentGateway->charge(100.00);
    return $success ? "Payment successful!" : "Payment failed!";
}
```

PROF

Method 2: Facade (Optional)

Create a Facade for easier access:

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

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

```
class Payment extends Facade
{
    protected static function getFacadeAccessor()
    {
        return \App\Services\PaymentGateway::class;
    }
}
```

Then use:

```
use App\Facades\Payment;

Payment::charge(100.00);
```

---

6. Testing

```
php artisan tinker
>>> app('App\Services\PaymentGateway')->charge(50.00); // Should return
`true`
```

Key Takeaways

✅ **Singleton Binding**: Ensures **one instance** per request.
✅ **Deferred Loading**: Optimizes performance by loading **only when needed**.
✅ **Dependency Injection**: Clean, testable way to use services.

Next Step: Try **Repository Pattern** for cleaner database interactions! 🚀

PROF

Implement the Repository Pattern with Eloquent and Dependency Injection

Objective

Learn how to:

1. **Decouple business logic from Eloquent** using the Repository Pattern.
2. **Use Dependency Injection (DI)** for cleaner, testable code.
3. **Swap data sources** (e.g., switch from MySQL to an API) without changing business logic.

---

1. Why Use the Repository Pattern?

✅ **Separation of concerns** (Business logic ≠ Database logic).
✅ **Easier testing** (Mock repositories instead of hitting the DB).
✅ **Flexibility** (Switch between Eloquent, APIs, or other data sources).

---

2. Project Structure

```
app/
├── Repositories/
│   ├── Interfaces/UserRepositoryInterface.php
│   ├── Eloquent/UserRepository.php
├── Providers/RepositoryServiceProvider.php
```

---

3. Step-by-Step Implementation

Step 1: Define the Repository Interface

```php
// app/Repositories/Interfaces/UserRepositoryInterface.php
namespace App\Repositories\Interfaces;

interface UserRepositoryInterface
{
    public function all();
    public function find(int $id);
    public function create(array $data);
    public function update(int $id, array $data);
    public function delete(int $id);
}
```

PROF

Step 2: Implement the Eloquent Repository

```php
// app/Repositories/Eloquent/UserRepository.php
namespace App\Repositories\Eloquent;

use App\Models\User;
use App\Repositories\Interfaces\UserRepositoryInterface;

class UserRepository implements UserRepositoryInterface
{
    protected $model;

    public function __construct(User $model)
    {
        $this->model = $model;
    }
    public function all()
    {
        return $this->model->all();
    }

    public function find(int $id)
    {
        return $this->model->findOrFail($id);
    }

    public function create(array $data)
    {
        return $this->model->create($data);
    }

    public function update(int $id, array $data)
    {
        $user = $this->model->findOrFail($id);
        $user->update($data);
        return $user;
    }

    public function delete(int $id)
    {
        return $this->model->destroy($id);
    }
}
```

Step 3: Bind Interface to Implementation (Service Provider)

```
// app/Providers/RepositoryServiceProvider.php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Repositories\Interfaces\UserRepositoryInterface;
use App\Repositories\Eloquent\UserRepository;

class RepositoryServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            UserRepositoryInterface::class,
            UserRepository::class
        );
    }

    public function boot()
    {
```

PROF

```
        // Optional: Bind other repositories here
    }
}
```

Register the provider in `config/app.php`:

```
'providers' => [
    // ...
    App\Providers\RepositoryServiceProvider::class,
],
```

4. Using the Repository in Controllers (Dependency Injection)

```
// app/Http/Controllers/UserController.php
namespace App\Http\Controllers;

use App\Repositories\Interfaces\UserRepositoryInterface;

class UserController extends Controller
{
    protected $userRepository;

    // Inject the repository interface
    public function __construct(UserRepositoryInterface $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function index()
    {
        $users = $this->userRepository->all();
        return view('users.index', compact('users'));
    }

    public function store(Request $request)
    {
        $data = $request->validate([...]);
        $user = $this->userRepository->create($data);
        return redirect()->route('users.index');
    }
}
```

PROF

5. Testing with Mock Repositories

```php
// tests/Feature/UserControllerTest.php
use App\Repositories\Interfaces\UserRepositoryInterface;
use Mockery;

test('index returns all users', function () {
    $mockRepo = Mockery::mock(UserRepositoryInterface::class);
    $mockRepo->shouldReceive('all')->once()->andReturn(collect([]));

    $this->app->instance(UserRepositoryInterface::class, $mockRepo);

    $response = $this->get('/users');
    $response->assertStatus(200);
});
```

6. Advanced: Extending for Multiple Models

1. **Create a `BaseRepository`** (optional for DRY code):

```php
// app/Repositories/Eloquent/BaseRepository.php
namespace App\Repositories\Eloquent;

use Illuminate\Database\Eloquent\Model;

class BaseRepository
{
    protected $model;

    public function __construct(Model $model)
    {
        $this->model = $model;
    }

    // Common methods (all, find, create, update, delete)
}
```

PROF

2. **Extend in `UserRepository`:**

```php
class UserRepository extends BaseRepository implements
UserRepositoryInterface
{
    public function __construct(User $model)
    {
        parent::__construct($model);
    }
    // Add custom methods (e.g., findByEmail)
}
```

Key Takeaways

✅ **Decouples business logic** from Eloquent.
✅ **Easy to test** (swap real DB for mock repositories).
✅ **Flexible** (switch to API/Redis storage later).

Set Up Authorization Using Policies and Gates for a Multi-Role User System

Objective

Learn how to:

1. **Use Policies** (for model-based authorization).
2. **Use Gates** (for general actions, like admin access).
3. **Manage multi-role permissions** (e.g., `admin`, `editor`, `user`).

---

1. Setup User Roles

First, add a `role` column to users:

```
php artisan make:migration add_role_to_users_table --table=users
```

```
// In the migration file
public function up()
{
    Schema::table('users', function (Blueprint $table) {
        $table->enum('role', ['admin', 'editor', 'user'])-
>default('user');
    });
}
```

PROF

Run:

```
php artisan migrate
```

---

2. Option 1: Using Policies (Model-Based Authorization)

Step 1: Generate a Policy

```
php artisan make:policy PostPolicy --model=Post
```

This creates:

```
// app/Policies/PostPolicy.php
namespace App\Policies;

use App\Models\User;
use App\Models\Post;

class PostPolicy
{
    public function viewAny(User $user) { return true; }
    public function view(User $user, Post $post) { return true; }
    public function create(User $user) { return $user->role === 'admin'
|| $user->role === 'editor'; }
    public function update(User $user, Post $post) { return $user->role
=== 'admin' || ($user->role === 'editor' && $post->user_id === $user-
>id); }
    public function delete(User $user, Post $post) { return $user->role
=== 'admin'; }
}
```

Step 2: Register the Policy

```
// app/Providers/AuthServiceProvider.php
protected $policies = [
    Post::class => PostPolicy::class,
];
```

PROF

Step 3: Use in Controllers

```
// app/Http/Controllers/PostController.php
public function edit(Post $post)
{
    $this->authorize('update', $post); // Checks PostPolicy::update()
    return view('posts.edit', compact('post'));
}
```

Or in **Blade views**:

```
@can('update', $post)
    <a href="{{ route('posts.edit', $post) }}">Edit</a>
@endcan
```

---

3. Option 2: Using Gates (General Actions)

Step 1: Define Gates

```
// app/Providers/AuthServiceProvider.php
public function boot()
{
    $this->registerPolicies();

    // Admin gate
    Gate::define('admin', function (User $user) {
        return $user->role === 'admin';
    });

    // Editor gate
    Gate::define('editor', function (User $user) {
        return $user->role === 'editor' || $user->role === 'admin';
    });
}
```

Step 2: Use in Controllers

```
public function dashboard()
{
    if (Gate::allows('admin')) {
        return view('admin.dashboard');
    }
    abort(403);
}
```

PROF

Or in **Blade views**:

```
@can('admin')
    <a href="/admin">Admin Panel</a>
@endcan
```

---

4. Combining Policies + Gates for Multi-Role Systems

Example: Allow admins to edit any post, editors only their own

```
// app/Policies/PostPolicy.php
public function update(User $user, Post $post)
{
    return $user->role === 'admin' ||
           ($user->role === 'editor' && $post->user_id === $user->id);
}
```

```
// app/Providers/AuthServiceProvider.php
Gate::define('manage-posts', function (User $user) {
    return $user->role === 'admin' || $user->role === 'editor';
});
```

Now, check in **Blade**:

```
@can('manage-posts')
    @can('update', $post)
        <a href="{{ route('posts.edit', $post) }}">Edit</a>
    @endcan
@endcan
```

---

5. Testing Authorization

```
// tests/Feature/PostPolicyTest.php
public function test_admin_can_update_any_post()
{
    $admin = User::factory()->create(['role' => 'admin']);
    $post = Post::factory()->create();

    $this->assertTrue($admin->can('update', $post));
}

public function test_editor_can_update_only_their_posts()
{
    $editor = User::factory()->create(['role' => 'editor']);
    $post = Post::factory()->create(['user_id' => $editor->id]);
    $otherPost = Post::factory()->create();

    $this->assertTrue($editor->can('update', $post));
    $this->assertFalse($editor->can('update', $otherPost));
}
```

Key Takeaways

✅ **Policies**: Best for **model-specific** permissions (e.g., `Post`, `User`).
✅ **Gates**: Best for **general actions** (e.g., `admin`, `editor` access).
✅ **Multi-Role Systems**: Combine both for granular control.

Creating a Custom Event and Listener for Welcome Emails

In this example, I'll show you how to create a custom event and listener in Laravel to send a welcome email when a user registers. This follows the event-driven architecture that Laravel provides.

Step 1: Create the Event

First, let's create the event that will be dispatched when a user registers.

```
// app/Events/UserRegistered.php

namespace App\Events;

use App\Models\User;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class UserRegistered
{
    use Dispatchable, SerializesModels;

    public $user;

    /**
     * Create a new event instance.
     *
     * @param User $user
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }
}
```

Step 2: Create the Listener

Next, create the listener that will handle sending the welcome email.

PROF

```
// app/Listeners/SendWelcomeEmail.php

namespace App\Listeners;

use App\Events\UserRegistered;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Facades\Mail;
use App\Mail\WelcomeEmail;

class SendWelcomeEmail implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param UserRegistered $event
     * @return void
     */
    public function handle(UserRegistered $event)
    {
        Mail::to($event->user->email)
            ->send(new WelcomeEmail($event->user));
    }
}
```

Step 3: Create the Welcome Email

Create a mailable class for the welcome email.

PROF

```
// app/Mail/WelcomeEmail.php

namespace App\Mail;

use App\Models\User;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class WelcomeEmail extends Mailable
{
    use Queueable, SerializesModels;

    public $user;

    /**
     * Create a new message instance.
     *
```

```
         * @param User $user
         */
        public function __construct(User $user)
        {
            $this->user = $user;
        }

        /**
         * Build the message.
         *
         * @return $this
         */
        public function build()
        {
            return $this->subject('Welcome to Our Application')
                        ->view('emails.welcome');
        }
    }
```

Step 4: Create the Email Template

Create a Blade template for the email.

```
// resources/views/emails/welcome.blade.php

<!DOCTYPE html>
<html>
<head>
    <title>Welcome Email</title>
</head>
<body>
    <h1>Welcome, {{ $user->name }}!</h1>
    <p>Thank you for registering with our application.</p>
    <p>We're excited to have you on board!</p>
</body>
</html>
```

PROF

Step 5: Register the Event and Listener

Add the event and listener to the `EventServiceProvider`.

```
// app/Providers/EventServiceProvider.php

namespace App\Providers;

use App\Events\UserRegistered;
use App\Listeners\SendWelcomeEmail;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
```

```
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        UserRegistered::class => [
            SendWelcomeEmail::class,
        ],
    ];

    // ...
}
```

Step 6: Dispatch the Event

Finally, dispatch the event when a user registers. This would typically be in your registration controller.

```
// app/Http/Controllers/Auth/RegisterController.php

namespace App\Http\Controllers\Auth;

use App\Events\UserRegistered;
use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;
use Illuminate\Foundation\Auth\RegistersUsers;

class RegisterController extends Controller
{
    use RegistersUsers;

    // ...

    protected function create(array $data)
    {
        $user = User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
        ]);

        // Dispatch the event
        event(new UserRegistered($user));

        return $user;
    }
}
```

Additional Notes

1. **Queueing**: The listener implements `ShouldQueue`, which means the email will be sent via a queue job. Make sure your queue worker is running.
2. **Testing**: You can test this by mocking the event or listener in your tests.
3. **Multiple Listeners**: You can add more listeners to the `UserRegistered` event if needed (e.g., for sending a Slack notification).
4. **Error Handling**: Consider adding error handling in the listener for cases where the email fails to send.

This implementation provides a clean separation of concerns and makes it easy to add additional actions when a user registers without modifying the registration logic itself.

Configuring Redis Queues and Laravel Horizon for Efficient Background Job Processing

This guide will walk you through setting up Redis as your queue driver and configuring Laravel Horizon for monitoring and managing your queue workers.

1. Install Required Packages

First, install the necessary packages via Composer:

```
composer require laravel/horizon predis/predis
```

PROF

2. Configure Redis as Queue Driver

Update your `.env` file to use Redis as the queue driver:

```
QUEUE_CONNECTION=redis

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

3. Configure Queue Connections

In `config/queue.php`, ensure your Redis connection is properly configured:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => env('REDIS_QUEUE', 'default'),
    'retry_after' => 90,
    'block_for' => null,
    'after_commit' => false,
],
```

4. Publish Horizon Assets

Publish Horizon's configuration and assets:

```
php artisan horizon:install
```

This will create:

- `config/horizon.php` (configuration file)
- `app/Providers/HorizonServiceProvider.php` (service provider)
- Dashboard view files

5. Configure Horizon

Edit `config/horizon.php` to configure your environments and worker settings:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default', 'notifications', 'emails'],
            'balance' => 'auto',
            'processes' => 10,
            'tries' => 3,
            'timeout' => 60,
            'memory' => 128,
        ],
    ],

    'local' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default'],
            'balance' => 'simple',
            'processes' => 3,
            'tries' => 3,
            'timeout' => 60,
            'memory' => 128,
```

PROF

```
            ],
        ],
    ],
```

6. Start Horizon

Run Horizon in your local environment:

```
php artisan horizon
```

For production, you should configure a process manager like Supervisor to keep Horizon running.

7. Configure Supervisor for Production (Optional but Recommended)

Create a Supervisor configuration file at `/etc/supervisor/conf.d/horizon.conf`:

```
[program:horizon]
process_name=%(program_name)s
command=php /path/to/your/project/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/path/to/your/project/storage/logs/horizon.log
stopwaitsecs=3600
```

Then update Supervisor:

PROF

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start horizon
```

8. Monitoring and Metrics

Horizon provides a dashboard at `/horizon` (protect this route in production). You can also configure metrics:

```
// config/horizon.php
'metrics' => [
    'trim_snapshots' => [
        'job' => 24,
        'queue' => 24,
        ],
    ],
```

9. Queue Worker Configuration Tips

Balancing Strategies:

- `simple`: Process jobs in order
- `auto`: Balance based on queue workload
- `false`: Process all queues equally

Memory Limits:

Set appropriate memory limits based on your job requirements:

```
'memory_limit' => 64, // MB
```

Timeouts:

Configure timeouts to prevent stuck jobs:

```
'timeout' => 60, // seconds
```

10. Testing Your Setup

1. Create a test job:

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

PROF

2. Dispatch the job:

```
ProcessPodcast::dispatch()->onQueue('default');
```

3. Check Horizon dashboard to see the job processing.

11. Performance Optimization

1. **Prefetching**: In `config/horizon.php`, adjust the `balance_maxshift` and `balance_cooldown` values to optimize worker distribution.

2. **Prioritize Queues**: Use queue priorities in your supervisor configuration:

```
'queue' => ['high', 'default', 'low'],
```

3. **Monitor Performance**: Use Horizon's metrics to identify bottlenecks and adjust worker counts accordingly.

12. Security Considerations

1. Protect the Horizon dashboard by adding authentication in `app/Providers/HorizonServiceProvider.php`:

```
protected function gate()
{
    Gate::define('viewHorizon', function ($user) {
        return in_array($user->email, [
            'admin@example.com',
        ]);
    });
}
```

2. Use HTTPS for the Horizon dashboard in production.

13. Troubleshooting

Common issues and solutions:

1. **Jobs stuck in pending**:
    - Check if Horizon is running
    - Verify Redis connection
    - Check for queue name mismatches

PROF

2. **High memory usage**:
    - Reduce the number of processes
    - Lower the memory limit
    - Optimize your jobs

3. **Failed jobs**:
    - Configure retries appropriately
    - Set up failed job logging

By following this configuration, you'll have a robust queue processing system that can handle background jobs efficiently while providing visibility into your queue operations through Horizon's dashboard.

Implementing File Uploads with S3 and Local Disk Fallback in Laravel

This implementation will allow your application to use Amazon S3 for file storage in production while falling back to local storage in development environments.

1. Install Required Packages

First, install the AWS SDK for PHP:

```
composer require league/flysystem-aws-s3-v3
```

2. Configure Filesystems

Update your `.env` file with S3 credentials:

```
FILESYSTEM_DISK=s3

AWS_ACCESS_KEY_ID=your-access-key-id
AWS_SECRET_ACCESS_KEY=your-secret-access-key
AWS_DEFAULT_REGION=your-region
AWS_BUCKET=your-bucket-name
AWS_URL=your-bucket-url
AWS_ENDPOINT= # Only needed if using a custom endpoint (like
DigitalOcean Spaces)
```

Configure `config/filesystems.php`:

PROF

```
'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path('app'),
        'throw' => false,
    ],

    'public' => [
        'driver' => 'local',
        'root' => storage_path('app/public'),
        'url' => env('APP_URL').'/storage',
        'visibility' => 'public',
        'throw' => false,
    ],

    's3' => [
        'driver' => 's3',
        'key' => env('AWS_ACCESS_KEY_ID'),
        'secret' => env('AWS_SECRET_ACCESS_KEY'),
        'region' => env('AWS_DEFAULT_REGION'),
        'bucket' => env('AWS_BUCKET'),
        'url' => env('AWS_URL'),
        'endpoint' => env('AWS_ENDPOINT'),
        'use_path_style_endpoint' => false,
        'throw' => false,
    ],
],

'default' => env('FILESYSTEM_DISK', 'local'),
```

3. Create a File Service

Create a service class to handle file operations with automatic fallback:

```
// app/Services/FileUploadService.php

namespace App\Services;

use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class FileUploadService
{
    public function store($file, $directory = 'uploads', $disk = null)
    {
        $disk = $disk ?? $this->getDefaultDisk();

        $filename = $this->generateFilename($file);

        $path = Storage::disk($disk)->putFileAs(
            $directory,
            $file,
            $filename
        );

        return [
            'path' => $path,
            'url' => Storage::disk($disk)->url($path),
            'disk' => $disk,
            'filename' => $filename,
        ];
    }

    public function getDefaultDisk()
    {
        // Use S3 in production, local otherwise
        return app()->environment('production') ? 's3' : 'public';
```

PROF

```
    }

    protected function generateFilename($file)
    {
        return Str::random(40) . '.' . $file->extension();
    }

    public function delete($path, $disk = null)
    {
        $disk = $disk ?? $this->getDefaultDisk();

        return Storage::disk($disk)->delete($path);
    }

    public function exists($path, $disk = null)
    {
        $disk = $disk ?? $this->getDefaultDisk();

        return Storage::disk($disk)->exists($path);
    }

    public function getUrl($path, $disk = null)
    {
        $disk = $disk ?? $this->getDefaultDisk();

        return Storage::disk($disk)->url($path);
    }
}
```

4. Create a Form Request for Validation

PROF

```
// app/Http/Requests/StoreFileRequest.php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rules\File;

class StoreFileRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'file' => [
                'required',
                File::types(['jpg', 'jpeg', 'png', 'gif', 'pdf', 'doc',
'docx'])
                    ->max(10 * 1024), // 10MB
            ],
        ];
    }
}
```

5. Create a Controller

```
// app/Http/Controllers/FileController.php

namespace App\Http\Controllers;

use App\Http\Requests\StoreFileRequest;
use App\Services\FileUploadService;

class FileController extends Controller
{
    protected $fileService;

    public function __construct(FileUploadService $fileService)
    {
        $this->fileService = $fileService;
    }

    public function store(StoreFileRequest $request)
    {
        $file = $request->file('file');

        $result = $this->fileService->store($file);

        return response()->json([
            'success' => true,
            'url' => $result['url'],
            'path' => $result['path'],
        ]);
    }

    public function destroy($path)
    {
        $deleted = $this->fileService->delete($path);

        return response()->json([
            'success' => $deleted,
        ]);
    }
}
```

6. Set Up Routes

```
// routes/web.php

use App\Http\Controllers\FileController;

Route::middleware(['auth'])->group(function () {
    Route::post('/files', [FileController::class, 'store']);
    Route::delete('/files/{path}', [FileController::class, 'destroy'])
        ->where('path', '.*');
});
```

7. Create a Frontend Upload Component (Example with Alpine.js)

```
<div x-data="fileUpload()">
    <input type="file" @change="handleFileChange"
accept="image/*,.pdf,.doc,.docx">
    <div x-show="uploading">Uploading...</div>
    <div x-show="error" x-text="error"></div>

    <template x-if="url">
        <div>
            <img x-show="isImage" :src="url" class="max-w-full h-auto">
            <a x-show="!isImage" :href="url" target="_blank">Download
File</a>
            <button @click="removeFile">Remove</button>
        </div>
    </template>
</div>

<script>
function fileUpload() {
    return {
        file: null,
        url: null,
        path: null,
        uploading: false,
        error: null,
        isImage: false,

        handleFileChange(e) {
            this.file = e.target.files[0];
            this.error = null;

            if (!this.file) return;

            this.uploadFile();
        },
```

PROF

```
        async uploadFile() {
            this.uploading = true;

            const formData = new FormData();
            formData.append('file', this.file);

            try {
                const response = await fetch('/files', {
                    method: 'POST',
                    body: formData,
                    headers: {
                        'X-CSRF-TOKEN':
document.querySelector('meta[name="csrf-token"]').content,
                    },
                });

                const data = await response.json();

                if (data.success) {
                    this.url = data.url;
                    this.path = data.path;
                    this.isImage = this.file.type.startsWith('image/');
                } else {
                    this.error = 'Upload failed';
                }
            } catch (err) {
                this.error = 'An error occurred';
            } finally {
                this.uploading = false;
            }
        },

        async removeFile() {
            if (!this.path) return;

            try {
                const response = await
fetch(`/files/${encodeURIComponent(this.path)}`, {
                    method: 'DELETE',
                    headers: {
                        'X-CSRF-TOKEN':
document.querySelector('meta[name="csrf-token"]').content,
                    },
                });

                const data = await response.json();

                if (data.success) {
                    this.url = null;
                    this.path = null;
                }
            } catch (err) {
                this.error = 'Failed to delete file';
```

PROF

```
                }
            }
        };
    }
    </script>
```

8. Configure CORS for S3 (If Needed)

If you're getting CORS errors when accessing files directly from S3:

1. Go to your S3 bucket → Permissions → CORS configuration
2. Add a policy like this:

```
[
    {
        "AllowedHeaders": ["*"],
        "AllowedMethods": ["GET", "HEAD"],
        "AllowedOrigins": ["https://yourdomain.com"],
        "ExposeHeaders": []
    }
]
```

9. Testing the Fallback Mechanism

To test the local fallback:

1. Set `FILESYSTEM_DISK=public` in your `.env` file
2. Create a symbolic link from `public/storage` to `storage/app/public`:

```
php artisan storage:link
```

PROF

10. Additional Considerations

1. **File Visibility**: For private files, use:

   ```
   Storage::disk('s3')->putFileAs($path, $file, 'private');
   ```

2. **Temporary URLs**: For private files, generate temporary URLs:

   ```
   Storage::disk('s3')->temporaryUrl($path, now()->addMinutes(30));
   ```

3. **File Size Limitations**: Adjust Nginx/Apache configuration if uploading large files:

```
; php.ini
upload_max_filesize = 20M
post_max_size = 20M
```

This implementation provides a robust file upload system that automatically uses S3 in production and falls back to local storage in development, with a clean service layer abstraction for easy maintenance.

Transforming API Responses with Eloquent API Resources and Pagination

Laravel's Eloquent API Resources provide a powerful way to transform your models and collections into JSON responses with full control over the structure. Here's how to implement them with pagination.

1. Create an API Resource

Generate a resource for your model (e.g., `User`):

```
php artisan make:resource UserResource
```

This creates `app/Http/Resources/UserResource.php`:

PROF

```
namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at->format('Y-m-d H:i:s'),
            'updated_at' => $this->updated_at->format('Y-m-d H:i:s'),

            // Relationships
            'posts' => PostResource::collection($this-
>whenLoaded('posts')),
            'profile' => new ProfileResource($this-
>whenLoaded('profile')),
        ];
    }
}
```

2. Create a Resource Collection (Optional)

For more control over collection responses:

```
php artisan make:resource UserCollection --collection
```

Or use the existing resource with `::collection()` method (shown below).

3. Implement Paginated API Endpoint

In your controller:

```
namespace App\Http\Controllers;

use App\Http\Resources\UserResource;
use App\Models\User;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function index(Request $request)
    {
        $users = User::with(['posts', 'profile'])
            ->filter($request->all())
            ->paginate($request->per_page ?? 15);

        return UserResource::collection($users);
    }

    public function show(User $user)
    {
        $user->load(['posts', 'profile']);
        return new UserResource($user);
    }
}
```

PROF

4. Customizing Pagination Response

To customize the pagination structure, create a custom resource collection:

```
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;
```

```
class UserCollection extends ResourceCollection
{
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'pagination' => [
                'total' => $this->total(),
                'count' => $this->count(),
                'per_page' => $this->perPage(),
                'current_page' => $this->currentPage(),
                'total_pages' => $this->lastPage(),
                'links' => [
                    'next' => $this->nextPageUrl(),
                    'previous' => $this->previousPageUrl(),
                ],
            ],
        ];
    }

    public function with($request)
    {
        return [
            'meta' => [
                'version' => '1.0',
                'api_status' => 'stable',
            ],
        ];
    }
}
```

Then update your controller to use the collection:

PROF

```
public function index(Request $request)
{
    $users = User::with(['posts', 'profile'])
        ->filter($request->all())
        ->paginate($request->per_page ?? 15);

    return new UserCollection($users);
}
```

5. Conditional Attributes and Relationships

Add logic to include fields conditionally:

```
// In UserResource
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->when($request->user()->isAdmin(), $this-
>email),
        'roles' => RoleResource::collection($this->whenLoaded('roles')),
        'profile' => new ProfileResource($this->whenLoaded('profile')),
        'created_at' => $this->when($request->show_timestamps, $this-
>created_at),
    ];
}
```

6. Adding Metadata to Responses

Add metadata at the resource level:

```
public function with($request)
{
    return [
        'meta' => [
            'version' => '1.0.0',
            'author' => 'Your API Team',
        ],
    ];
}
```

7. Example API Response

PROF

With pagination, the response will look like:

```
{
    "data": [
        {
            "id": 1,
            "name": "John Doe",
            "email": "john@example.com",
            "created_at": "2023-05-15 10:00:00",
            "posts": [
                {
                    "id": 1,
                    "title": "First Post"
                }
            ],
            "profile": {
                "bio": "Web developer"
            }
        }
    ],
    "links": {
        "first": "http://example.com/users?page=1",
        "last": "http://example.com/users?page=5",
        "prev": null,
        "next": "http://example.com/users?page=2"
    },
    "meta": {
        "current_page": 1,
        "from": 1,
        "last_page": 5,
        "path": "http://example.com/users",
        "per_page": 15,
        "to": 15,
        "total": 75,
        "version": "1.0",
        "api_status": "stable"
    }
}
```

8. Sorting and Filtering

Enhance your endpoint with sorting and filtering:

```
public function index(Request $request)
{
    $query = User::query()->with(['posts', 'profile']);

    // Filtering
    if ($request->has('name')) {
        $query->where('name', 'like', '%' . $request->name . '%');
    }

    // Sorting
    $sortField = $request->sort_field ?? 'created_at';
    $sortOrder = $request->sort_order ?? 'desc';
    $query->orderBy($sortField, $sortOrder);

    // Pagination
    $perPage = $request->per_page ?? 15;
    $users = $query->paginate($perPage);

    return new UserCollection($users);
}
```

PROF

9. API Resource Best Practices

1. **Keep resources lean** - Only include necessary data
2. **Use relationships sparingly** - Load them explicitly with `with()`
3. **Consistent structure** - Maintain the same response format across endpoints
4. **Document your API** - Use tools like Swagger/OpenAPI
5. **Version your API** - Include version in response metadata

10. Testing Your API Resources

Create tests to ensure your resources work as expected:

```php
namespace Tests\Feature;

use App\Http\Resources\UserResource;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class UserResourceTest extends TestCase
{
    use RefreshDatabase;

    public function test_user_resource_structure()
    {
        $user = User::factory()->create();

        $resource = new UserResource($user);

        $response = $resource->response()->getData(true);

        $this->assertArrayHasKey('data', $response);
        $this->assertEquals([
            'id',
            'name',
            'email',
            'created_at',
            'updated_at',
        ], array_keys($response['data']));
    }

    public function test_user_collection_pagination()
    {
        User::factory()->count(20)->create();

        $response = $this->getJson('/api/users?per_page=5');

        $response->assertJsonStructure([
            'data',
            'links',
            'meta' => [
                'current_page',
                'per_page',
```

PROF

```
                'total',
            ],
        ]);
    }
}
```

This implementation gives you full control over your API responses while maintaining clean, maintainable code. The pagination integration ensures your API is scalable and follows RESTful best practices.

Creating Custom Form Request Validation with Dynamic Rules and Messages

Laravel's Form Request validation provides a powerful way to organize your validation logic. Here's how to create dynamic validation with customizable rules and error messages.

1. Generate a Form Request

First, create a custom Form Request class:

```
php artisan make:request DynamicFormRequest
```

2. Implement Dynamic Rules

Modify the generated file to include dynamic rules:

PROF

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class DynamicFormRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true; // Or add your authorization logic
    }

    /**
     * Get the validation rules that apply to the request.
     */
    public function rules(): array
    {
        $baseRules = [
            'email' => ['required', 'email', 'max:255'],
            'name' => ['required', 'string', 'max:100'],
        ];

        // Add conditional rules
        if ($this->has('update_profile')) {
            $baseRules['avatar'] = ['nullable', 'image', 'max:2048'];
            $baseRules['bio'] = ['nullable', 'string', 'max:500'];
        }

        // Dynamic role-based rules
        if ($this->user()->isAdmin()) {
            $baseRules['role'] = ['required', Rule::in(['admin',
'editor', 'user'])];
        }

        // Merge with any additional rules from the request
        return array_merge($baseRules, $this->additionalRules());
    }

    /**
     * Generate additional rules based on request data
     */
    protected function additionalRules(): array
    {
        $additional = [];

        // Example: Add validation for dynamic fields
        if ($this->has('custom_fields')) {
            foreach ($this->input('custom_fields') as $field => $value)
{
                $additional["custom_fields.{$field}"] = $this-
>getFieldRule($field);
            }
        }

        return $additional;
    }

    /**
     * Get validation rule for a specific dynamic field
     */
    protected function getFieldRule(string $field): array
    {
        // You might get these from a database or config
        $fieldRules = [
            'age' => ['numeric', 'min:18', 'max:120'],
            'website' => ['nullable', 'url'],
            'phone' => ['required', 'regex:/^\+?[0-9]{10,15}$/'],
        ];
```

PROF

```
        return $fieldRules[$field] ?? ['string', 'max:255'];
    }
}
```

3. Customize Error Messages

Add dynamic error messages to the same class:

```
/**
 * Get custom error messages for validator errors.
 */
public function messages(): array
{
    return [
        // General messages
        'required' => 'The :attribute field is required.',
        'email' => 'The :attribute must be a valid email address.',

        // Field-specific messages
        'name.required' => 'Please provide your full name.',
        'name.max' => 'Your name cannot exceed 100 characters.',

        // Dynamic field messages
        'custom_fields.*.numeric' => 'The :attribute must be a number.',
        'custom_fields.age.min' => 'You must be at least 18 years old.',

        // Conditional messages
        'role.required' => $this->user()->isAdmin()
            ? 'Please select a user role.'
            : 'You cannot set roles.',
    ];
}
```

PROF

4. Add Custom Attributes

Improve the display names of fields in error messages:

```
/**
 * Get custom attributes for validator errors.
 */
public function attributes(): array
{
    $attributes = [
        'email' => 'email address',
        'custom_fields.age' => 'age',
    ];
```

```
        // Add dynamic field attributes
        if ($this->has('custom_fields')) {
            foreach (array_keys($this->input('custom_fields')) as $field) {
                $attributes["custom_fields.{$field}"] = str_replace('_', '
', $field);
            }
        }

        return $attributes;
    }
```

5. Using the Form Request in a Controller

Here's how to use your dynamic form request:

```
namespace App\Http\Controllers;

use App\Http\Requests\DynamicFormRequest;

class UserController extends Controller
{
    public function updateProfile(DynamicFormRequest $request)
    {
        // The request is already validated at this point

        $validated = $request->validated();

        // Process the validated data
        auth()->user()->update($validated);

        return response()->json([
            'message' => 'Profile updated successfully',
            'data' => $validated,
        ]);
    }
}
```

PROF

6. Advanced: Dynamic Rules Based on Database

For rules that depend on database values:

```
protected function additionalRules(): array
{
    $additional = [];

    // Get validation rules from database configuration
    $validationConfig = \App\Models\ValidationConfig::where('model',
'User')->get();
```

```
    foreach ($validationConfig as $config) {
        $additional[$config->field_name] = explode('|', $config-
>validation_rules);
    }

    return $additional;
}
```

7. Handling Array Validation

For complex array data validation:

```
public function rules(): array
{
    return [
        'products' => ['required', 'array'],
        'products.*.id' => ['required', 'exists:products,id'],
        'products.*.quantity' => ['required', 'integer', 'min:1'],
        'products.*.options' => ['sometimes', 'array'],
        'products.*.options.*' => ['string', 'max:50'],
    ];
}

public function messages(): array
{
    return [
        'products.*.id.exists' => 'One or more products are invalid.',
        'products.*.quantity.min' => 'Quantity must be at least 1 for
all items.',
    ];
}
```

PROF

8. Testing Your Dynamic Validation

Create tests to verify your validation logic:

```
namespace Tests\Feature\Http\Requests;

use App\Http\Requests\DynamicFormRequest;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Support\Facades\Validator;
use Tests\TestCase;

class DynamicFormRequestTest extends TestCase
{
    use RefreshDatabase;
    public function test_validation_rules_are_dynamic()
    {
        $request = new DynamicFormRequest();

        // Simulate request with custom fields
        $request->merge([
            'email' => 'test@example.com',
            'name' => 'John Doe',
            'custom_fields' => [
                'age' => 25,
                'website' => 'https://example.com'
            ]
        ]);

        $validator = Validator::make(
            $request->all(),
            $request->rules(),
            $request->messages(),
            $request->attributes()
        );

        $this->assertFalse($validator->fails());
    }

    public function test_admin_role_validation()
    {
        $admin = \App\Models\User::factory()->admin()->create();
        $this->actingAs($admin);

        $request = new DynamicFormRequest();
        $request->merge([
            'email' => 'admin@example.com',
            'name' => 'Admin User',
            'role' => 'editor'
        ]);

        $rules = $request->rules();

        $this->assertArrayHasKey('role', $rules);
        $this->assertContains('required', $rules['role']);
    }
}
```

PROF

9. Additional Tips

1. **Rule Objects**: Create custom rule objects for complex validation logic
2. **After Hooks**: Use `withValidator` to add after-validation logic
3. **Prepare for Validation**: Use `prepareForValidation` to modify data before validation

Example with `withValidator`:

```
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with
this field');
        }
    });
}
```

This implementation gives you complete control over your validation logic while keeping it organized and maintainable. The dynamic nature allows you to adapt to different scenarios without creating multiple request classes.

Implementing Database Multi-tenancy with Separate Databases per Tenant

This solution provides a comprehensive approach to multi-tenancy where each tenant has their own dedicated database, ensuring complete data isolation.

1. Setup Tenant Identification

First, establish how to identify tenants (typically via subdomain or request header):

```
// app/Providers/TenancyServiceProvider.php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Config;

class TenancyServiceProvider extends ServiceProvider
{
    public function register()
    {
        // Detect tenant from subdomain
        $host = request()->getHost();
        $subdomain = explode('.', $host)[0];

        // Or from header (for API requests)
        $tenantId = request()->header('X-Tenant-ID') ?? $subdomain;

        // Store tenant identifier
        Config::set('tenant.id', $tenantId);
    }
```

PROF

```php
    public function boot()
    {
        // Register middleware to handle tenant switching
        $this->app['router']->aliasMiddleware('tenant',
\App\Http\Middleware\SetTenantDatabase::class);
    }
}
```

Don't forget to register this provider in `config/app.php`.

2. Create Tenant Database Middleware

```php
// app/Http/Middleware/SetTenantDatabase.php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Config;
use Illuminate\Support\Facades\DB;

class SetTenantDatabase
{
    public function handle($request, Closure $next)
    {
        $tenantId = Config::get('tenant.id');

        if (!$tenantId) {
            abort(403, 'Tenant not identified');
        }

        // Verify tenant exists and get database config
        $tenant = \App\Models\Tenant::where('identifier', $tenantId)-
>firstOrFail();

        // Configure the tenant's database
        Config::set('database.connections.tenant', [
            'driver' => 'mysql',
            'host' => $tenant->db_host,
            'port' => $tenant->db_port,
            'database' => $tenant->db_name,
            'username' => $tenant->db_username,
            'password' => $tenant->db_password,
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'prefix_indexes' => true,
            'strict' => true,
            'engine' => null,
        ]);
```

PROF

```
        // Set as default connection
        DB::purge('tenant');
        Config::set('database.default', 'tenant');
        DB::reconnect('tenant');

        return $next($request);
    }
}
```

3. Tenant Model and Migration

Create a model and migration for storing tenant information:

```
php artisan make:model Tenant -m
```

```
// database/migrations/xxxx_create_tenants_table.php

public function up()
{
    Schema::create('tenants', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('identifier')->unique(); // subdomain or tenant
ID
        $table->string('db_host');
        $table->string('db_name');
        $table->string('db_username');
        $table->string('db_password');
        $table->integer('db_port')->default(3306);
        $table->json('meta')->nullable();
        $table->timestamps();
    });
}
```

4. Central Database Configuration

Configure your central database (for tenant records) in `.env`:

```
DB_CONNECTION=central
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=central_db
DB_USERNAME=root
DB_PASSWORD=
```

Add the connection to `config/database.php`:

```
'connections' => [
    'central' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', '127.0.0.1'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'central_db'),
        'username' => env('DB_USERNAME', 'root'),
        'password' => env('DB_PASSWORD', ''),
        'charset' => 'utf8mb4',
        'collation' => 'utf8mb4_unicode_ci',
        'prefix' => '',
        'prefix_indexes' => true,
        'strict' => true,
        'engine' => null,
    ],
    // ... other connections
],
```

5. Tenant Database Creation Command

Create an Artisan command to provision new tenant databases:

```
php artisan make:command CreateTenant
```

```
// app/Console/Commands/CreateTenant.php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use App\Models\Tenant;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Str;

class CreateTenant extends Command
{
    protected $signature = 'tenant:create {name} {--subdomain=}';
    protected $description = 'Create a new tenant with dedicated
database';

    public function handle()
    {
        $name = $this->argument('name');
        $subdomain = $this->option('subdomain') ?? Str::slug($name);
        // Create database
        $dbName = 'tenant_' . Str::random(8);
        $dbUsername = 'tenant_' . Str::random(8);
        $dbPassword = Str::random(16);

        // Create database and user (MySQL example)
        DB::connection('central')->statement("CREATE DATABASE
{$dbName}");
        DB::connection('central')->statement("CREATE USER
'{$dbUsername}'@'%' IDENTIFIED BY '{$dbPassword}'");
        DB::connection('central')->statement("GRANT ALL PRIVILEGES ON
{$dbName}.* TO '{$dbUsername}'@'%'");
        DB::connection('central')->statement("FLUSH PRIVILEGES");

        // Create tenant record
        $tenant = Tenant::create([
            'name' => $name,
            'identifier' => $subdomain,
            'db_host' => env('DB_HOST', '127.0.0.1'),
            'db_name' => $dbName,
            'db_username' => $dbUsername,
            'db_password' => $dbPassword,
            'db_port' => env('DB_PORT', 3306),
        ]);

        // Run migrations for the new tenant
        $this->migrateTenantDatabase($tenant);

        $this->info("Tenant {$name} created successfully!");
        $this->info("Subdomain: {$subdomain}");
        $this->info("Database: {$dbName}");
    }

    protected function migrateTenantDatabase(Tenant $tenant)
    {
        config([
            'database.connections.tenant.database' => $tenant->db_name,
            'database.connections.tenant.username' => $tenant-
>db_username,
            'database.connections.tenant.password' => $tenant-
>db_password,
        ]);

        DB::purge('tenant');

        $this->call('migrate', [
            '--database' => 'tenant',
            '--path' => 'database/migrations/tenant',
            '--force' => true,
        ]);
    }
}
```

PROF

6. Tenant-Specific Migrations

Create tenant-specific migrations in a separate directory:

```
mkdir database/migrations/tenant
```

Example tenant migration:

```
// database/migrations/tenant/xxxx_create_tenant_tables.php

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

    // Add other tenant-specific tables
}
```

7. Tenant Model Trait

Create a trait to make models tenant-aware:

PROF

```
// app/Traits/TenantScoped.php

namespace App\Traits;

trait TenantScoped
{
    public static function bootTenantScoped()
    {
        static::addGlobalScope(new TenantScope);
    }
}
```

And the scope:

```
// app/Scopes/TenantScope.php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class TenantScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        // No need for tenant_id column since we're using separate
databases
        // This is just here for consistency with other multi-tenancy
approaches
    }
}
```

8. Route Middleware

Apply tenant middleware to routes:

```
// routes/web.php

Route::middleware(['tenant'])->group(function () {
    // All tenant-specific routes
    Route::get('/', [TenantController::class, 'dashboard']);

    // Other tenant routes...
});
```

PROF

9. Tenant Switch Controller

For admin users who need to switch between tenants:

```
// app/Http/Controllers/TenantSwitchController.php

namespace App\Http\Controllers;

use App\Models\Tenant;
use Illuminate\Support\Facades\Session;

class TenantSwitchController extends Controller
{
    public function switch(Tenant $tenant)
    {
        if (!auth()->user()->isSuperAdmin()) {
            abort(403);
        }

        Session::put('tenant_id', $tenant->identifier);

        return redirect()->to('http://' . $tenant->identifier . '.' .
config('app.domain'));
    }
}
```

10. Testing the Implementation

Create tests for your multi-tenancy setup:

```
// tests/Feature/TenancyTest.php

namespace Tests\Feature;

use App\Models\Tenant;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class TenancyTest extends TestCase
{
    use RefreshDatabase;

    protected $tenant;

    protected function setUp(): void
    {
        parent::setUp();

        $this->tenant = Tenant::create([
            'name' => 'Test Tenant',
            'identifier' => 'test',
            'db_host' => env('DB_HOST', '127.0.0.1'),
            'db_name' => 'tenant_test',
            'db_username' => 'tenant_test',
            'db_password' => 'password',
            'db_port' => env('DB_PORT', 3306),
        ]);

        // Setup tenant database (in memory for testing)
        config([
            'database.connections.tenant.database' => ':memory:',
            'database.default' => 'tenant',
        ]);

        $this->artisan('migrate', ['--database' => 'tenant']);
```

PROF

```
    }

    public function test_tenant_database_isolation()
    {
        // Create a record in tenant database
        \App\Models\User::create(['name' => 'Tenant User', 'email' =>
'user@tenant.com', 'password' => 'secret']);

        // Verify it exists in tenant DB
        $this->assertDatabaseHas('users', ['email' =>
'user@tenant.com']);

        // Switch to central DB
        config(['database.default' => 'central']);

        // Verify it doesn't exist in central DB
        $this->assertDatabaseMissing('users', ['email' =>
'user@tenant.com']);
    }
}
```

11. Deployment Considerations

1. **Database Hosting**: Consider using cloud database services that support easy provisioning
2. **Backups**: Implement a backup strategy for all tenant databases
3. **Migrations**: Create a system to apply migrations across all tenant databases
4. **Monitoring**: Set up monitoring for all databases
5. **Connection Pooling**: Configure connection pooling for better performance

12. Scaling Considerations

1. **Database Sharding**: Distribute tenant databases across multiple servers
2. **Read Replicas**: Implement read replicas for high-traffic tenants
3. **Caching**: Use Redis or Memcached with tenant-specific prefixes
4. **Queue Workers**: Configure separate queue workers per tenant if needed

PROF

This implementation provides complete database isolation for each tenant while maintaining a manageable central system for tenant administration. The solution is scalable and can be adapted to various hosting environments.

Feature Tests for CRUD Operations with Authentication

Here's a comprehensive guide to writing PHPUnit feature tests for CRUD operations with authentication in Laravel.

1. Setup Test Environment

First, ensure your `phpunit.xml` is properly configured:

```
<php>
    <env name="APP_ENV" value="testing"/>
    <env name="DB_CONNECTION" value="sqlite"/>
    <env name="DB_DATABASE" value=":memory:"/>
    <env name="SESSION_DRIVER" value="array"/>
    <env name="QUEUE_CONNECTION" value="sync"/>
</php>
```

2. Test Case Base Class

Create a base test case class for authentication:

```
// tests/Feature/AuthenticatedTestCase.php

namespace Tests\Feature;

use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

abstract class AuthenticatedTestCase extends TestCase
{
    use RefreshDatabase, WithFaker;

    protected $user;
    protected $authToken;

    protected function setUp(): void
    {
        parent::setUp();

        // Create and authenticate a test user
        $this->user = User::factory()->create();
        $this->authToken = $this->user->createToken('test-token')-
>plainTextToken;
    }

    protected function authenticatedJson($method, $uri, array $data =
[], array $headers = [])
    {
        $headers['Authorization'] = 'Bearer ' . $this->authToken;
        return $this->json($method, $uri, $data, $headers);
    }
}
```

PROF

3. User Registration Test

```
// tests/Feature/Auth/RegistrationTest.php

namespace Tests\Feature\Auth;

use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class RegistrationTest extends TestCase
{
    use RefreshDatabase;

    public function test_new_users_can_register()
    {
        $response = $this->postJson('/api/register', [
            'name' => 'Test User',
            'email' => 'test@example.com',
            'password' => 'password',
            'password_confirmation' => 'password',
        ]);

        $response->assertStatus(201)
            ->assertJsonStructure([
                'data' => [
                    'user' => [
                        'id',
                        'name',
                        'email',
                    ],
                    'access_token'
                ]
            ]);

        $this->assertDatabaseHas('users', [
            'email' => 'test@example.com'
        ]);
    }

    public function test_registration_validation()
    {
        // Test required fields
        $response = $this->postJson('/api/register', []);
        $response->assertStatus(422)
            ->assertJsonValidationErrors(['name', 'email', 'password']);

        // Test password confirmation
        $response = $this->postJson('/api/register', [
            'name' => 'Test User',
```

PROF

```
            'email' => 'test@example.com',
            'password' => 'password',
            'password_confirmation' => 'wrong',
        ]);
        $response->assertStatus(422)
            ->assertJsonValidationErrors(['password']);

        // Test unique email
        User::factory()->create(['email' => 'test@example.com']);
        $response = $this->postJson('/api/register', [
            'name' => 'Test User',
            'email' => 'test@example.com',
            'password' => 'password',
            'password_confirmation' => 'password',
        ]);
        $response->assertStatus(422)
            ->assertJsonValidationErrors(['email']);
    }
}
```

4. Authentication Test

```
// tests/Feature/Auth/AuthenticationTest.php

namespace Tests\Feature\Auth;

use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class AuthenticationTest extends TestCase
{
    use RefreshDatabase;

    public function test_users_can_authenticate()
    {
        $user = User::factory()->create();

        $response = $this->postJson('/api/login', [
            'email' => $user->email,
            'password' => 'password',
        ]);

        $response->assertStatus(200)
            ->assertJsonStructure([
                'data' => [
                    'access_token',
                    'token_type'
                ]
            ]);
```

PROF

```
    }

    public function
test_users_can_not_authenticate_with_invalid_password()
    {
        $user = User::factory()->create();

        $response = $this->postJson('/api/login', [
            'email' => $user->email,
            'password' => 'wrong-password',
        ]);

        $response->assertStatus(401)
            ->assertJson(['message' => 'Invalid credentials']);
    }

    public function test_authenticated_user_can_get_their_profile()
    {
        $user = User::factory()->create();
        $token = $user->createToken('test-token')->plainTextToken;

        $response = $this->withHeaders([
            'Authorization' => 'Bearer ' . $token,
        ])->getJson('/api/me');

        $response->assertStatus(200)
            ->assertJson([
                'data' => [
                    'email' => $user->email
                ]
            ]);
    }

    public function test_users_can_logout()
    {
        $user = User::factory()->create();
        $token = $user->createToken('test-token')->plainTextToken;

        $response = $this->withHeaders([
            'Authorization' => 'Bearer ' . $token,
        ])->postJson('/api/logout');

        $response->assertStatus(200)
            ->assertJson(['message' => 'Successfully logged out']);

        $this->assertCount(0, $user->tokens);
    }
}
```

PROF

5. CRUD Operations Test

Assuming we have a Post model with CRUD operations:

```
// tests/Feature/PostCrudTest.php

namespace Tests\Feature;

use App\Models\Post;
use Tests\Feature\AuthenticatedTestCase;

class PostCrudTest extends AuthenticatedTestCase
{
    public function test_user_can_create_post()
    {
        $postData = [
            'title' => 'Test Post',
            'content' => 'This is a test post content',
        ];

        $response = $this->authenticatedJson('POST', '/api/posts',
$postData);

        $response->assertStatus(201)
            ->assertJsonStructure([
                'data' => [
                    'id',
                    'title',
                    'content',
                    'user_id',
                    'created_at',
                ]
            ])
            ->assertJson([
                'data' => [
                    'title' => 'Test Post',
                    'user_id' => $this->user->id,
                ]
            ]);

        $this->assertDatabaseHas('posts', [
            'title' => 'Test Post',
            'user_id' => $this->user->id,
        ]);
    }

    public function test_user_can_view_their_posts()
    {
        $post = Post::factory()->create(['user_id' => $this->user->id]);

        $response = $this->authenticatedJson('GET', '/api/posts');

        $response->assertStatus(200)
            ->assertJsonStructure([
```

PROF

```
                'data' => [
                    '*' => [
                        'id',
                        'title',
                        'content',
                        'user_id',
                    ]
                ]
            ])
            ->assertJsonFragment([
                'id' => $post->id,
                'title' => $post->title,
            ]);
    }

    public function test_user_can_view_single_post()
    {
        $post = Post::factory()->create(['user_id' => $this->user->id]);

        $response = $this->authenticatedJson('GET', "/api/posts/{$post-
>id}");

        $response->assertStatus(200)
            ->assertJson([
                'data' => [
                    'id' => $post->id,
                    'title' => $post->title,
                    'content' => $post->content,
                ]
            ]);
    }

    public function test_user_can_update_their_post()
    {
        $post = Post::factory()->create(['user_id' => $this->user->id]);

        $updateData = [
            'title' => 'Updated Title',
            'content' => 'Updated content',
        ];

        $response = $this->authenticatedJson('PUT', "/api/posts/{$post-
>id}", $updateData);

        $response->assertStatus(200)
            ->assertJson([
                'data' => [
                    'id' => $post->id,
                    'title' => 'Updated Title',
                    'content' => 'Updated content',
                ]
            ]);
```

PROF

```
        $this->assertDatabaseHas('posts', [
            'id' => $post->id,
            'title' => 'Updated Title',
        ]);
    }

    public function test_user_can_delete_their_post()
    {
        $post = Post::factory()->create(['user_id' => $this->user->id]);

        $response = $this->authenticatedJson('DELETE',
"/api/posts/{$post->id}");

        $response->assertStatus(204);

        $this->assertDatabaseMissing('posts', [
            'id' => $post->id,
        ]);
    }

    public function test_user_cannot_access_other_users_posts()
    {
        $otherUser = User::factory()->create();
        $post = Post::factory()->create(['user_id' => $otherUser->id]);

        // Try to view
        $response = $this->authenticatedJson('GET', "/api/posts/{$post-
>id}");
        $response->assertStatus(403);

        // Try to update
        $response = $this->authenticatedJson('PUT', "/api/posts/{$post-
>id}", [
            'title' => 'Unauthorized Update',
        ]);
        $response->assertStatus(403);

        // Try to delete
        $response = $this->authenticatedJson('DELETE',
"/api/posts/{$post->id}");
        $response->assertStatus(403);

        // Verify no changes were made
        $this->assertDatabaseHas('posts', [
            'id' => $post->id,
            'title' => $post->title,
        ]);
    }

    public function test_validation_rules_for_post_operations()
    {
        // Create validation
        $response = $this->authenticatedJson('POST', '/api/posts', []);
        $response->assertStatus(422)
            ->assertJsonValidationErrors(['title', 'content']);

        // Update validation
        $post = Post::factory()->create(['user_id' => $this->user->id]);
        $response = $this->authenticatedJson('PUT', "/api/posts/{$post-
>id}", []);
        $response->assertStatus(422)
            ->assertJsonValidationErrors(['title', 'content']);
    }
}
```

6. Testing Invalid Authentication

```
// tests/Feature/Auth/InvalidAuthenticationTest.php

namespace Tests\Feature\Auth;

use Tests\TestCase;

class InvalidAuthenticationTest extends TestCase
{
    public function
test_unauthenticated_users_cannot_access_protected_routes()
    {
        // Try to access protected route without token
        $response = $this->getJson('/api/me');
        $response->assertStatus(401);

        // Try with invalid token
        $response = $this->withHeaders([
            'Authorization' => 'Bearer invalid-token',
        ])->getJson('/api/me');
        $response->assertStatus(401);
    }

    public function test_invalid_token_format()
    {
        $response = $this->withHeaders([
            'Authorization' => 'InvalidFormat',
        ])->getJson('/api/me');
        $response->assertStatus(401);
    }
}
```

PROF

7. Running the Tests

Run your tests with:

```
php artisan test
```

Or run specific test classes:

```
php artisan test --filter RegistrationTest
php artisan test --filter PostCrudTest
```

8. Additional Testing Tips

1. **Test Coverage**: Add `@covers` annotations to document what each test covers
2. **Data Providers**: Use data providers for testing multiple input scenarios
3. **Database Transactions**: Consider using `DatabaseTransactions` instead of `RefreshDatabase` for faster tests
4. **Factories**: Create comprehensive factories for all your models
5. **HTTP Tests**: Use Laravel's HTTP test helpers for API testing

Example with data provider:

```
/**
 * @dataProvider invalidRegistrationDataProvider
 */
public function test_registration_validation($input, $errors)
{
    $response = $this->postJson('/api/register', $input);
    $response->assertStatus(422)
        ->assertJsonValidationErrors($errors);
}

public function invalidRegistrationDataProvider()
{
    return [
        'missing name' => [
            ['email' => 'test@test.com', 'password' => 'password',
'password_confirmation' => 'password'],
            ['name']
        ],
        'invalid email' => [
            ['name' => 'Test', 'email' => 'not-an-email', 'password' =>
'password', 'password_confirmation' => 'password'],
            ['email']
        ],
        'password mismatch' => [
            ['name' => 'Test', 'email' => 'test@test.com', 'password' =>
'password', 'password_confirmation' => 'different'],
            ['password']
        ],
```

PROF

```
    ];
}
```

These tests provide comprehensive coverage for CRUD operations with authentication, ensuring your API endpoints are secure and function as expected.

Dockerizing a Laravel App with PHP, MySQL, and Nginx

Here's a complete guide to dockerizing a Laravel application using Docker Compose with PHP, MySQL, and Nginx.

1. Project Structure

First, let's create the following project structure:

```
your-laravel-app/
├── docker/
│   ├── nginx/
│   │   └── default.conf
│   └── php/
│       └── Dockerfile
├── src/ (your Laravel application files)
├── docker-compose.yml
└── .env
```

2. Docker Compose File

Create a `docker-compose.yml` file:

```
version: '3.8'

services:
  app:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_app
    restart: unless-stopped
    working_dir: /var/www
    volumes:
      - ./src:/var/www
    environment:
      - DB_HOST=mysql
      - DB_DATABASE=${DB_DATABASE}
      - DB_USERNAME=${DB_USERNAME}
      - DB_PASSWORD=${DB_PASSWORD}
    depends_on:
      - mysql

  webserver:
    image: nginx:alpine
    container_name: laravel_webserver
    restart: unless-stopped
    ports:
      - "8000:80"
    volumes:
      - ./src:/var/www
      - ./docker/nginx/default.conf:/etc/nginx/conf.d/default.conf

  mysql:
    image: mysql:8.0
    container_name: laravel_mysql
    restart: unless-stopped
    environment:
      - MYSQL_DATABASE=${DB_DATABASE}
      - MYSQL_USER=${DB_USERNAME}
      - MYSQL_PASSWORD=${DB_PASSWORD}
      - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}
    volumes:
      - mysql_data:/var/lib/mysql
    ports:
      - "3306:3306"

volumes:
  mysql_data:
    driver: local
```

3. PHP Dockerfile

PROF

Create `docker/php/Dockerfile`:

```
FROM php:8.2-fpm-alpine

WORKDIR /var/www

RUN apk update && apk add \
    build-base \
    vim \
    curl \
    php82 \
    php82-fpm \
    php82-common \
    php82-pdo \
    php82-pdo_mysql \
    php82-mysqli \
    php82-mcrypt \
    php82-mbstring \
    php82-xml \
    php82-openssl \
    php82-json \
    php82-phar \
    php82-zip \
    php82-gd \
    php82-dom \
    php82-session \
    php82-zlib

RUN docker-php-ext-install pdo pdo_mysql mysqli

RUN apk add --no-cache zip libzip-dev
RUN docker-php-ext-configure zip
RUN docker-php-ext-install zip

RUN apk add --no-cache jpeg-dev libpng-dev freetype-dev
RUN docker-php-ext-configure gd --with-freetype --with-jpeg
RUN docker-php-ext-install gd

RUN curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer

RUN addgroup -g 1000 -S www && \
    adduser -u 1000 -S www -G www

USER www

COPY --chown=www:www . /var/www

EXPOSE 9000
CMD ["php-fpm"]
```

PROF

4. Nginx Configuration

Create `docker/nginx/default.conf`:

```
server {
    listen 80;
    index index.php index.html;
    server_name localhost;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/public;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass app:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

5. Environment Variables

Create or update your `.env` file (both in root and src directory):

```
DB_CONNECTION=mysql
DB_HOST=mysql
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=laravel_user
DB_PASSWORD=secret
```

6. Build and Run the Containers

1. First, make sure you have Docker installed and running on your system.

2. Build and start the containers:

PROF

```
docker-compose up -d --build
```

3. Install Laravel dependencies (run inside the app container):

```
docker-compose exec app composer install
```

4. Generate application key:

```
docker-compose exec app php artisan key:generate
```

5. Run migrations (if needed):

```
docker-compose exec app php artisan migrate
```

7. Access the Application

Your Laravel application should now be accessible at:

- Web: http://localhost:8000
- MySQL: localhost:3306 (use credentials from .env)

Additional Commands

- To stop the containers:

  ```
  docker-compose down
  ```

- To stop and remove volumes (including database data):

  ```
  docker-compose down -v
  ```

- To view logs:

  ```
  docker-compose logs -f
  ```

This setup provides a complete development environment for Laravel with PHP 8.2, MySQL 8.0, and Nginx. You can customize the versions and configurations as needed for your specific project requirements.

PROF

Real-Time Notifications in Laravel with WebSockets and Pusher

Here's a comprehensive guide to implementing real-time notifications in your Laravel application using either Laravel WebSockets (self-hosted) or Pusher (cloud-based).

Option 1: Using Laravel WebSockets (Self-Hosted)

1. Install Required Packages

```
composer require beyondcode/laravel-websockets
composer require pusher/pusher-php-server
```

2. Publish Configurations

```
php artisan vendor:publish --
provider="BeyondCode\LaravelWebSockets\WebSocketsServiceProvider" --
tag="config"
php artisan vendor:publish --
provider="BeyondCode\LaravelWebSockets\WebSocketsServiceProvider" --
tag="migrations"
```

3. Configure Environment Variables (.env)

```
BROADCAST_DRIVER=pusher

PUSHER_APP_ID=12345
PUSHER_APP_KEY=your-app-key
PUSHER_APP_SECRET=your-app-secret
PUSHER_APP_CLUSTER=mt1

# For Laravel WebSockets
LARAVEL_WEBSOCKETS_PORT=6001
```

4. Configure config/broadcasting.php

```
'connections' => [
    'pusher' => [
        'driver' => 'pusher',
        'key' => env('PUSHER_APP_KEY'),
        'secret' => env('PUSHER_APP_SECRET'),
        'app_id' => env('PUSHER_APP_ID'),
        'options' => [
            'cluster' => env('PUSHER_APP_CLUSTER'),
            'encrypted' => true,
            'host' => '127.0.0.1',
            'port' => env('LARAVEL_WEBSOCKETS_PORT', 6001),
            'scheme' => 'http',
            'useTLS' => false,
        ],
    ],
],
```

PROF

5. Create Notification Event

```
php artisan make:event NotificationEvent
```

Update the event:

```php
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class NotificationEvent implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $message;
    public $userId;

    public function __construct($message, $userId)
    {
        $this->message = $message;
        $this->userId = $userId;
    }

    public function broadcastOn()
    {
        return new Channel('user.'.$this->userId);
    }

    public function broadcastAs()
    {
        return 'notification.event';
    }
}
```

PROF

6. Create Notification Component (Frontend)

Install Laravel Echo and Pusher JS:

```
npm install --save laravel-echo pusher-js
```

Add to `resources/js/bootstrap.js`:

```javascript
import Echo from 'laravel-echo';
```

```
window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    wsHost: window.location.hostname,
    wsPort: 6001,
    forceTLS: false,
    disableStats: true,
    enabledTransports: ['ws', 'wss'],
});
```

7. Listen for Events in JavaScript

```
// Listen for notifications for the authenticated user
const userId = document.querySelector("meta[name='user-id']").content;

window.Echo.private(`user.${userId}`)
    .listen('.notification.event', (data) => {
        console.log('New notification:', data.message);
        // Update UI with the new notification
        showNotification(data.message);
    });

function showNotification(message) {
    // Implement your notification UI logic here
    alert(message); // Simple example
}
```

8. Trigger Notifications from Your Code

```
// Example controller method
public function sendNotification()
{
    $message = "This is a real-time notification!";
    $userId = auth()->id(); // Or any user ID you want to notify

    event(new NotificationEvent($message, $userId));

    return response()->json(['status' => 'Notification sent!']);
}
```

9. Run WebSockets Server

```
php artisan websockets:serve
```

10. Update Docker Setup (if using Docker)

Add this to your `docker-compose.yml`:

```
websocket:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_websocket
    command: php artisan websockets:serve
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
```

Option 2: Using Pusher (Cloud Service)

1. Install Pusher Package

```
composer require pusher/pusher-php-server
```

2. Configure Environment Variables (.env)

```
BROADCAST_DRIVER=pusher

PUSHER_APP_ID=your-app-id
PUSHER_APP_KEY=your-app-key
PUSHER_APP_SECRET=your-app-secret
PUSHER_APP_CLUSTER=your-cluster
```

3. Update config/broadcasting.php

```
'options' => [
    'cluster' => env('PUSHER_APP_CLUSTER'),
    'encrypted' => true,
    'useTLS' => true,
],
```

4. Frontend Setup

Update `resources/js/bootstrap.js`:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

5. No Need to Run WebSocket Server

Pusher handles the WebSocket server for you in the cloud.

Testing Your Implementation

1. Start your Laravel application and WebSocket server (if using Laravel WebSockets)
2. Open two browser windows/tabs logged in as the same user
3. Trigger a notification from one window (via your controller method)
4. The other window should receive the notification in real-time

Additional Features

Database Notifications

Combine with Laravel's database notifications:

```
php artisan notifications:table
php artisan migrate
```

Create a notification:

```
php artisan make:notification RealTimeNotification
```

Update the notification:

```
public function toArray($notifiable)
{
    return [
        'message' => $this->message,
    ];
}

public function toBroadcast($notifiable)
{
    return new NotificationEvent($this->message, $notifiable->id);
}
```

Presence Channels (for online users)

```
public function broadcastOn()
{
    return new PresenceChannel('notifications.'.$this->userId);
}
```

Frontend:

```
window.Echo.join(`notifications.${userId}`)
    .here((users) => {
        console.log('Online users:', users);
    })
    .joining((user) => {
        console.log('User joined:', user);
    })
    .leaving((user) => {
        console.log('User left:', user);
    });
```

This implementation provides a complete real-time notification system that you can further customize based on your application's requirements.

Integrating GraphQL with Laravel using Lighthouse

Here's a comprehensive guide to setting up GraphQL in your Laravel application using the Lighthouse package, which provides a powerful and flexible GraphQL implementation.

PROF

1. Installation

Install required packages:

```
composer require nuwave/lighthouse
```

Publish the Lighthouse configuration:

```
php artisan vendor:publish --
provider="Nuwave\Lighthouse\LighthouseServiceProvider" --tag="config"
```

2. Configuration

Update your `.env` file:

```
LIGHTHOUSE_CACHE_ENABLE=false # Set to true in production
```

Configure `config/lighthouse.php` (optional):

```
return [
    'schema' => [
        'register' => base_path('graphql/schema.graphql'),
    ],
    // Other configuration options...
];
```

3. Set Up GraphQL Schema

Create a `graphql/schema.graphql` file in your project root:

```
type Query {
    users: [User!]! @all
    user(id: ID @eq): User @find
    posts: [Post!]! @all
    post(id: ID @eq): Post @find
}

type Mutation {
    createUser(name: String!, email: String! @rules(apply: ["email",
"unique:users"]), password: String! @hash): User @create
    updateUser(id: ID!, name: String, email: String): User @update
    deleteUser(id: ID!): User @delete
}

type User {
    id: ID!
    name: String!
    email: String!
    posts: [Post!]! @hasMany
    created_at: DateTime!
    updated_at: DateTime!
}

type Post {
    id: ID!
    title: String!
    content: String!
```

PROF

```
    author: User! @belongsTo
    created_at: DateTime!
    updated_at: DateTime!
}
```

4. Create Models and Migrations

User model (app/Models/User.php):

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    protected $fillable = [
        'name',
        'email',
        'password',
    ];

    protected $hidden = [
        'password',
        'remember_token',
    ];

    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

PROF

Post model (app/Models/Post.php):

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

```php
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use HasFactory;

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

    public function author()
    {
        return $this->belongsTo(User::class, 'user_id');
    }
}
```

Migration for posts table:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('content');
        $table->foreignId('user_id')->constrained();
        $table->timestamps();
    });
}
```

PROF

Run migrations:

```
php artisan migrate
```

5. Set Up GraphQL Controller (Optional)

Create a controller to handle GraphQL requests:

```
php artisan make:controller GraphQLController
```

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class GraphQLController extends Controller
{
    public function __invoke(Request $request)
    {
        return response()->json(
            app('graphql')->executeQuery(
                $request->input('query'),
                null,
                [
                    'request' => $request,
                ],
                $request->input('variables')
            )->toArray()
        );
    }
}
```

Add route in `routes/api.php`:

```
use App\Http\Controllers\GraphQLController;

Route::post('/graphql', GraphQLController::class);
```

6. Testing GraphQL API

Using GraphQL Playground

PROF

Install GraphQL Playground:

```
composer require mll-lab/laravel-graphql-playground
```

Access it at `/graphql-playground` in your browser.

Example Queries:

Query all users with their posts:

```
query {
  users {
    id
    name
    email
    posts {
      id
      title
    }
  }
}
```

Create a new user:

```
mutation {
  createUser(
    name: "John Doe"
    email: "john@example.com"
    password: "secret"
  ) {
    id
    name
    email
  }
}
```

Update a user:

```
mutation {
  updateUser(
    id: 1
    name: "Updated Name"
    email: "updated@example.com"
  ) {
    id
    name
    email
  }
}
```

PROF

7. Advanced Features

Custom Resolvers

Create a custom resolver for complex queries:

```
php artisan make:graphql:query CustomUserQuery
```

Update the generated file:

```
<?php

namespace App\GraphQL\Queries;

use App\Models\User;

class CustomUserQuery
{
    public function __invoke($rootValue, array $args, $context,
$resolveInfo)
    {
        return User::where('name', 'LIKE', "%{$args['search']}%")
            ->with('posts')
            ->get();
    }
}
```

Add to your schema:

```
type Query {
    searchUsers(search: String!): [User!]! @field(resolver:
"App\\GraphQL\\Queries\\CustomUserQuery")
}
```

Authentication

Add authentication middleware in `config/lighthouse.php`:

PROF

```
'middleware' => [
    \Illuminate\Cookie\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,

\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::clas
s,
    'api',
],
```

Create a login mutation:

```
type Mutation {
    login(email: String!, password: String!): String!
}
```

Create a resolver:

```
php artisan make:graphql:mutation Login
```

```php
<?php

namespace App\GraphQL\Mutations;

use Illuminate\Support\Facades\Auth;

class Login
{
    public function __invoke($rootValue, array $args, $context,
$resolveInfo)
    {
        if (Auth::attempt($args)) {
            $user = Auth::user();
            return $user->createToken('graphql')->plainTextToken;
        }

        throw new \Exception('Invalid credentials');
    }
}
```

Authorization

Use directives for authorization:

PROF

```
type Mutation {
    deletePost(id: ID!): Post @delete @can(ability: "delete", find:
"id")
}
```

Create a policy:

```
php artisan make:policy PostPolicy --model=Post
```

8. Performance Optimization

Enable Schema Caching (Production)

```
php artisan lighthouse:cache
```

Pagination

```
type Query {
    postsPaginated(page: Int = 1, count: Int = 15): [Post!]!
@paginate(type: "paginator" model: "App\\Models\\Post")
}
```

DataLoader for N+1 Problem

Lighthouse automatically batches queries to avoid N+1 problems.

9. Docker Integration

Update your `docker-compose.yml` to ensure GraphQL works:

```
services:
  app:
    # ... existing config ...
    ports:
      - "8000:8000"
      - "8001:8001" # For GraphQL subscriptions if needed
```

10. Testing with PHPUnit

Create a test:

PROF

```
php artisan make:test GraphQLTest
```

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Models\User;

class GraphQLTest extends TestCase
{
    public function testUserQuery()
    {
        $user = User::factory()->create();
        $response = $this->postJson('/graphql', [
            'query' => '{
                user(id: '.$user->id.') {
                    id
                    name
                    email
                }
            }'
        ]);

        $response->assertStatus(200)
            ->assertJson([
                'data' => [
                    'user' => [
                        'id' => (string) $user->id,
                        'name' => $user->name,
                        'email' => $user->email,
                    ]
                ]
            ]);
    }
}
```

This implementation provides a complete GraphQL API with Laravel using Lighthouse, including queries, mutations, authentication, authorization, and testing. You can extend it further based on your application's specific requirements.

Optimizing Laravel Database Queries with Redis Caching

PROF

Redis is an excellent solution for caching frequently accessed data in your Laravel application. Here's a comprehensive guide to implementing Redis caching for database optimization.

1. Setup and Configuration

Install Required Packages

```
composer require predis/predis
```

Configure Environment Variables (.env)

```
CACHE_DRIVER=redis
REDIS_HOST=redis
```

```
REDIS_PASSWORD=null
REDIS_PORT=6379
```

Update Docker Compose

Add Redis service to your `docker-compose.yml`:

```
services:
  redis:
    image: redis:alpine
    container_name: laravel_redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    networks:
      - laravel_network

volumes:
  redis_data:
    driver: local
```

2. Basic Caching Implementation

Cache Database Results

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

// Get users with caching
$users = Cache::remember('users.all', now()->addHours(24), function () {
    return User::with('posts')->get();
});

// Get single user with caching
$user = Cache::remember('user.'.$id, now()->addHours(12), function ()
use ($id) {
    return User::with('posts')->findOrFail($id);
});
```

PROF

Cache GraphQL Resolvers

```
// In your GraphQL resolver
public function resolve($root, array $args, $context, $resolveInfo)
{
    return Cache::remember('custom_query.'.md5(json_encode($args)),
now()->addHour(), function () use ($args) {
        return User::where('name', 'LIKE', "%{$args['search']}%")
            ->with('posts')
            ->get();
    });
}
```

3. Advanced Caching Strategies

Model Caching with Observers

Create an observer:

```
php artisan make:observer UserObserver --model=User
```

Implement caching logic:

```
namespace App\Observers;

use App\Models\User;
use Illuminate\Support\Facades\Cache;

class UserObserver
{
    public function saved(User $user)
    {
        Cache::forget('users.all');
        Cache::forget('user.'.$user->id);
    }

    public function deleted(User $user)
    {
        Cache::forget('users.all');
        Cache::forget('user.'.$user->id);
    }
}
```

PROF

Register the observer in `AppServiceProvider`:

```
public function boot()
{
    User::observe(UserObserver::class);
}
```

Cache Tags for Complex Relationships

```
// Cache users with their posts using tags
$users = Cache::tags(['users', 'posts'])->remember('users.with_posts',
now()->addDay(), function () {
    return User::with('posts')->get();
});

// Clear all cache related to users
Cache::tags('users')->flush();
```

Database Query Cache Middleware

Create middleware:

```
php artisan make:middleware CacheResponses
```

Implement caching:

```
public function handle($request, Closure $next, $ttl = 60)
{
    $key = 'route.'.md5($request->url().serialize($request->all()));

    return Cache::remember($key, now()->addMinutes($ttl), function ()
use ($request, $next) {
        return $next($request);
    });
}
```

Register middleware in `Kernel.php`:

PROF

```
protected $routeMiddleware = [
    'cache.response' => \App\Http\Middleware\CacheResponses::class,
];
```

Use in routes:

```
Route::get('/users', 'UserController@index')-
>middleware('cache.response:1440'); // Cache for 24h
```

4. Real-Time Cache Invalidation

Event-Based Cache Clearing

```
// In your EventServiceProvider
protected $listen = [
    'App\Events\UserUpdated' => [
        'App\Listeners\ClearUserCache',
    ],
];

// Create listener
php artisan make:listener ClearUserCache
```

```
public function handle(UserUpdated $event)
{
    Cache::forget('user.'.$event->user->id);
    Cache::tags(['users'])->flush();
}
```

Automatic Cache Invalidation with Model Events

```
// In your model
protected static function boot()
{
    parent::boot();

    static::updated(function ($model) {
        Cache::tags([$model->getTable()])->flush();
    });

    static::deleted(function ($model) {
        Cache::tags([$model->getTable()])->flush();
    });
}
```

PROF

5. Performance Monitoring

Cache Hit/Miss Tracking

```
// Wrap your cache calls to track performance
$start = microtime(true);
$result = Cache::remember('key', $ttl, function () {
    // expensive operation
});
$elapsed = microtime(true) - $start;

Log::debug('Cache operation', [
    'key' => 'key',
    'time' => $elapsed,
    'hit' => Cache::has('key') // Before the call would be more accurate
]);
```

Redis-Specific Optimizations

```
// Pipeline multiple cache operations
Cache::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i, now()->addHour());
    }
});

// Use Redis hashes for related data
Cache::hmset('user:1:profile', [
    'name' => 'John',
    'email' => 'john@example.com'
]);
```

6. Testing Cache Implementation

Unit Test for Cached Queries

```
public function testUserIndexUsesCache()
{
    Cache::shouldReceive('remember')
        ->once()
        ->with('users.all', \Mockery::type(\DateTimeInterface::class),
\Mockery::type('Closure'))
        ->andReturn(collect([factory(User::class)->make()]));

    $response = $this->get('/users');
    $response->assertStatus(200);
}

public function testCacheInvalidationOnUserUpdate()
{
    $user = User::factory()->create();

    Cache::shouldReceive('forget')
        ->once()
        ->with('user.'.$user->id);

    Cache::shouldReceive('forget')
        ->once()
        ->with('users.all');
    $user->update(['name' => 'New Name']);
}
```

7. Production Considerations

Cache Stampede Protection

```
// Using atomic locks to prevent cache stampede
$value = Cache::lock('expensive_operation')->block(5, function () {
    return Cache::remember('expensive_data', now()->addHour(), function
() {
        // Expensive operation
    });
});
```

Multi-Layered Caching

```
// Check in-memory cache first, then Redis, then database
$users = $this->memoryCache->get('users.all', function () {
    return Cache::remember('users.all', now()->addHour(), function () {
        return User::all();
    });
});
```

Monitoring with Horizon

Configure Horizon to monitor Redis:

```
composer require laravel/horizon
php artisan horizon:install
```

Example Implementation for GraphQL

```
// Optimized GraphQL resolver with caching
public function resolve($root, array $args, $context, $resolveInfo)
{
    $cacheKey = 'graphql.'.md5($resolveInfo-
>fieldName.json_encode($args));

    return Cache::remember($cacheKey, now()->addHour(), function () use
($args) {
        return User::query()
            ->when(isset($args['search']), function ($query) use ($args)
{
                return $query->where('name', 'like', "%
{$args['search']}%");
            })
            ->with(['posts' => function ($query) {
                $query->select('id', 'title', 'user_id');
            }])
            ->get();
    });
}
```

Cache Configuration Tips

1. **Tiered Caching**: Use different TTLs for different data types
    - User profiles: 24 hours
    - Product listings: 1 hour
    - Shopping cart: 5 minutes
2. **Size Management**: Monitor Redis memory usage

```
redis-cli info memory
```

3. **Compression**: For large datasets, consider compression

```
Cache::put('large_data', gzcompress($data), $seconds);
```

By implementing these Redis caching strategies, you'll significantly reduce database load while improving response times for frequently accessed data. Remember to balance cache freshness with performance benefits based on your application's specific requirements.

PROF

Comprehensive Guide to Debugging with Laravel Telescope

Laravel Telescope is an elegant debug assistant that provides insight into requests, exceptions, jobs, queries, and more. Here's how to leverage it effectively in your Dockerized Laravel application.

1. Installation & Setup

Install Telescope in your Laravel app:

```
composer require laravel/telescope
```

Publish assets and run migrations:

```
php artisan telescope:install
php artisan migrate
```

Update your `.env`:

```
TELESCOPE_ENABLED=true
```

2. Docker Integration

Add Telescope service to your `docker-compose.yml`:

```
services:
  # ... existing services ...

  telescope:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_telescope
    command: php artisan telescope:watch
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
      - redis
```

PROF

3. Key Telescope Features Deep Dive

Request Monitoring

Telescope records all incoming requests with:

- Full request/response payloads
- Headers
- Session data
- User authentication status
- Timeline of application events

Example use case: Identify slow requests by sorting by duration.

Database Queries

Telescope shows:

- All executed SQL queries
- Bindings (click "Show bindings")
- Query duration
- Duplicate queries
- Slow queries (highlighted in red)

Pro tip: Click the "Explain" button to see the query execution plan.

Jobs & Queues

Monitor:

- Job payloads
- Status (pending, processed, failed)
- Execution time
- Exceptions
- Retry attempts

Example debugging:

```php
// Tag important jobs for easier filtering
dispatch(new ProcessPodcast($podcast))->tags(['audio-processing']);
```

Scheduled Tasks

View:

- Command output
- Runtime duration
- Exit code
- Next run time

Mail Preview

Intercept and inspect:

- Raw email content
- Headers
- Attachments
- Recipients

Notifications

See:

- Notification channels used
- Recipients
- Notification content

Cache Operations

Monitor:

- Cache hits/misses
- Stored values
- Tags usage
- Forget operations

Redis Commands

Track all Redis operations with:

- Command type (GET, SET, etc.)
- Keys accessed
- Execution time

4. Advanced Configuration

Customize `config/telescope.php`:

```
return [
    'storage' => [
        'database' => [
            'connection' => env('DB_TELESCOPE_CONNECTION', 'mysql'),
            'chunk' => 1000,
        ],
    ],

    'watchers' => [
        Watchers\CacheWatcher::class => [
            'enabled' => env('TELESCOPE_CACHE_WATCHER', true),
            'hidden' => [
                'password',
                'token',
            ],
        ],
        // Other watchers...
    ],

    'ignore_paths' => [
        'nova-api*',
        'telescope*',
    ],
];
```

Filter Sensitive Data

```
Telescope::filter(function (IncomingEntry $entry) {
    if ($entry->type === 'request') {
        return !Str::is('*/admin/*', $entry->content['uri']);
    }

    return true;
});
```

5. Practical Debugging Workflows

Debugging Slow Requests

1. Open Telescope dashboard
2. Sort requests by duration
3. Click into slow request
4. Analyze:
    - Database queries tab
    - Timeline tab
    - Redis operations

Identifying N+1 Problems

1. Look for repeated similar queries
2. Check query counts in the "Queries" tab
3. Use the "Duplicate Queries" filter

Job Failure Analysis

1. Navigate to "Jobs" section
2. Filter by failed status
3. Examine:
    - Exception stack trace
    - Job payload
    - Timeline of attempts

PROF

Memory Leak Detection

1. Monitor memory usage in request details
2. Look for steadily increasing values
3. Correlate with specific queries/jobs

6. Telescope in Production

Prune old entries:

```
php artisan telescope:prune --hours=48
```

Set up pruning automatically:

```
// In app/Console/Kernel.php
protected function schedule(Schedule $schedule)
{
    $schedule->command('telescope:prune --hours=48')->daily();
}
```

Limit access with authorization:

```
// In AppServiceProvider
Telescope::auth(function ($request) {
    return in_array($request->user()->email, [
        'admin@example.com'
    ]);
});
```

7. Integrating with Other Tools

Log Aggregation

```
Telescope::tag(function (IncomingEntry $entry) {
    return $entry->type === 'request'
        ? ['status:'.$entry->content['response_status']]
        : [];
});
```

PROF

Slack Notifications for Errors

```
// In AppServiceProvider
Telescope::night();

// Or custom callback
Telescope::filterExceptions(function ($entry) {
    return $entry->isReportableException();
});
```

8. Custom Watchers

Create a custom watcher for your domain:

```
php artisan make:watcher DeploymentWatcher
```

```
namespace App\Watchers;

use Laravel\Telescope\Telescope;
use Laravel\Telescope\Watchers\Watcher;

class DeploymentWatcher extends Watcher
{
    public function register($app)
    {
        $app->events->listen('deployment.started', [$this,
'recordDeployment']);
    }

    public function recordDeployment($deployment)
    {
        Telescope::recordDeployment(DeploymentEntry::make($deployment));
    }
}
```

Register in `config/telescope.php`:

```
'watchers' => [
    App\Watchers\DeploymentWatcher::class,
    // ...
]
```

PROF

9. Performance Considerations

For high-traffic apps:

```
// Sample only a percentage of requests
Telescope::filter(function (IncomingEntry $entry) {
    return random_int(0, 100) <= 20; // Sample 20% of requests
});
```

Use separate database:

```
DB_TELESCOPE_CONNECTION=telescope
```

Configure in `config/database.php`:

```
'telescope' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => 'telescope',
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    // ...
],
```

10. Security Best Practices

1. Always restrict access in production:

   ```
   Telescope::auth(function ($request) {
       return app()->environment('local') ||
              $request->user()->isAdmin();
   });
   ```

2. Filter sensitive data:

   ```
   Telescope::hideRequestParameters([
       'password',
       'credit_card',
   ]);
   ```

3. Use HTTPS for Telescope routes in production

Example Debugging Session

Problem: API endpoint responding slowly

1. Open Telescope dashboard
2. Filter to the problematic route
3. Examine the request:
   - Note high memory usage (247MB)
   - See 42 database queries
4. Switch to Queries tab:
   - Identify duplicate queries loading user relationships
5. Solution:

```
// Eager load relationships
return User::with(['posts', 'comments'])->get();
```

Telescope provides unparalleled visibility into your Laravel application's inner workings. By leveraging its comprehensive monitoring capabilities, you can quickly identify and resolve performance bottlenecks, debug errors, and understand your application's behavior in depth.

Implementing Full-Text Search with Laravel Scout and Algolia

Here's a comprehensive guide to setting up powerful full-text search in your Laravel application using Laravel Scout with Algolia.

1. Installation & Setup

Install required packages:

```
composer require laravel/scout
composer require algolia/algoliasearch-client-php
```

Publish Scout configuration:

```
php artisan vendor:publish --
provider="Laravel\Scout\ScoutServiceProvider"
```

Configure environment variables (.env):

```
SCOUT_DRIVER=algolia
ALGOLIA_APP_ID=your_app_id
ALGOLIA_SECRET=your_admin_api_key
ALGOLIA_SEARCH=your_search_only_api_key
```

2. Model Configuration

Prepare your model for indexing:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize the data array...
        $array['author_name'] = $this->author->name;
        $array['category_name'] = $this->category->name;

        return $array;
    }

    /**
     * Determine if the model should be searchable.
     */
    public function shouldBeSearchable()
    {
        return $this->is_published;
    }
}
```

PROF

3. Algolia Configuration

Create a custom index configuration:

```php
// In your model
public function searchableAs()
{
    return 'posts_index';
}
```

Configure index settings:

```php
// Typically in a service provider
use Algolia\AlgoliaSearch\SearchClient;
use Laravel\Scout\EngineManager;

public function boot()
{
    resolve(EngineManager::class)->extend('algolia', function () {
        $client = SearchClient::create(
            config('scout.algolia.id'),
            config('scout.algolia.secret')
        );

        // Custom index settings
        $index = $client->initIndex('posts_index');
        $index->setSettings([
            'searchableAttributes' => [
                'title',
                'content',
                'author_name',
                'category_name'
            ],
            'customRanking' => [
                'desc(published_at)',
                'desc(views_count)'
            ],
            'attributesForFaceting' => [
                'category_name',
                'author_name'
            ],
            'replicas' => [
                'posts_index_price_asc',
                'posts_index_price_desc'
            ],
        ]);

        return new AlgoliaEngine($client);
    });
}
```

PROF

4. Indexing Data

Import existing records:

```
php artisan scout:import "App\Models\Post"
```

Queueing imports (for large datasets):

```
php artisan queue:work
php artisan scout:import "App\Models\Post" --queue
```

Controlling chunk size:

```
php artisan scout:import "App\Models\Post" --chunk=100
```

5. Searching Implementation

Basic search in controller:

```
public function search(Request $request)
{
    return Post::search($request->input('query'))->get();
}
```

Advanced search with filters:

```
$results = Post::search($query)
    ->where('category_id', $categoryId)
    ->where('status', 'published')
    ->orderBy('published_at', 'desc')
    ->paginate(15);
```

Faceted search:

PROF

```
$results = Post::search($query)
    ->with([
        'filters' => 'category_name:Technology OR
category_name:Science',
        'facets' => ['category_name', 'author_name']
    ])
    ->paginate(15);
```

6. Frontend Integration

Install Algolia JavaScript client:

```
npm install algoliasearch instantsearch.js
```

Create search component:

```
import algoliasearch from 'algoliasearch/lite';
import instantsearch from 'instantsearch.js';

const searchClient = algoliasearch(
  process.env.MIX_ALGOLIA_APP_ID,
  process.env.MIX_ALGOLIA_SEARCH
);

const search = instantsearch({
  indexName: 'posts_index',
  searchClient,
});

search.addWidgets([
  instantsearch.widgets.searchBox({
    container: '#searchbox',
  }),

  instantsearch.widgets.hits({
    container: '#hits',
    templates: {
      item: `
        <div>
          <h3>{{#helpers.highlight}}{ "attribute": "title" }
{{/helpers.highlight}}</h3>
          <p>{{#helpers.highlight}}{ "attribute": "content" }
{{/helpers.highlight}}</p>
          <p>By: {{author_name}}</p>
        </div>
      `,
    },
  }),

  instantsearch.widgets.refinementList({
    container: '#category-list',
    attribute: 'category_name',
  }),
]);

search.start();
```

PROF

7. Real-Time Syncing

Handle model events:

```
// In your model
protected static function boot()
{
    parent::boot();

    static::created(function ($model) {
        $model->searchable();
    });

    static::updated(function ($model) {
        $model->searchable();
    });

    static::deleted(function ($model) {
        $model->unsearchable();
    });
}
```

Queue sync operations:

```
// In config/scout.php
'queue' => [
    'connection' => 'redis',
    'queue' => 'scout',
],
```

8. Performance Optimization

Selective indexing:

PROF

```
public function toSearchableArray()
{
    // Only index necessary fields
    return [
        'title' => $this->title,
        'content' => $this->content,
        'author' => $this->author->name,
        'published_at' => $this->published_at->timestamp,
    ];
}
```

Conditional indexing:

```
public function shouldBeSearchable()
{
    return $this->is_published && $this->approved;
}
```

9. Advanced Features

Synonyms:

```
$index->setSettings([
    'synonyms' => [
        ['objectID' => 'tech', 'type' => 'synonym', 'synonyms' =>
['tech', 'technology']],
        ['objectID' => 'mobile', 'type' => 'synonym', 'synonyms' =>
['mobile', 'cellphone', 'smartphone']]
    ]
]);
```

Typo tolerance:

```
$index->setSettings([
    'typoTolerance' => [
        'minWordSizeForTypos' => [
            'oneTypo' => 4,
            'twoTypos' => 8
        ]
    ]
]);
```

PROF

Geo-search:

```
// Add to your model
public function toSearchableArray()
{
    return [
        // ... other fields
        '_geoloc' => [
            'lat' => $this->latitude,
            'lng' => $this->longitude,
        ],
    ];
}

// Search usage
```

```
$results = Post::search($query)
    ->aroundLatLng($lat, $lng)
    ->within('5km')
    ->get();
```

10. Monitoring & Maintenance

Set up webhook for monitoring:

```
// In a service provider
Algolia::setSettings('posts_index', [
    'forwardToReplicas' => true,
    'attributesForFaceting' => ['filterOnly(category)'],
    'webhook' => [
        'url' => route('algolia.webhook'),
        'events' => ['settingsUpdate']
    ]
]);
```

Handle Algolia webhooks:

```
Route::post('/algolia/webhook', function (Request $request) {
    Log::info('Algolia settings updated', $request->all());
    return response()->json(['status' => 'success']);
})->middleware('auth:api');
```

11. Docker Integration

Update your `docker-compose.yml`:

PROF

```
services:
  # ... existing services ...

  scout:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_scout
    command: php artisan scout:flush "App\Models\Post" && php artisan
scout:import "App\Models\Post"
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
      - redis
```

12. Testing

Mock Scout in tests:

```
// In your test
Post::shouldReceive('search')
    ->once()
    ->with('test query')
    ->andReturnSelf();

Post::shouldReceive('get')
    ->once()
    ->andReturn(collect([Post::factory()->make()]));

$response = $this->get('/search?query=test+query');
$response->assertStatus(200);
```

Integration test:

```
public function test_search_returns_results()
{
    $post = Post::factory()->create(['title' => 'Test Post']);

    // Manually sync with Algolia
    $post->searchable();

    // Wait for indexing to complete
    sleep(1);

    $response = $this->get('/search?query=Test');

    $response->assertStatus(200)
        ->assertSee('Test Post');
}
```

PROF

This implementation provides a complete full-text search solution with Laravel Scout and Algolia, covering installation, configuration, advanced features, and maintenance. The system will provide fast, relevant search results with features like typo-tolerance, faceting, and geo-search.

Automating Tasks with Laravel Scheduler and Custom Commands

Laravel provides a powerful task scheduling system that allows you to automate various operations. Here's how to implement it effectively with cron jobs and custom commands in your Dockerized environment.

1. Setting Up the Scheduler

Create the Scheduler Kernel

All scheduled tasks are defined in `app/Console/Kernel.php`:

```
protected function schedule(Schedule $schedule)
{
    // Run database backups daily at midnight
    $schedule->command('db:backup')
             ->daily()
             ->onSuccess(function () {
                 Log::info('Database backup completed successfully');
             })
             ->onFailure(function () {
                 Log::error('Database backup failed');
             });

    // Process queued jobs every minute
    $schedule->command('queue:work --stop-when-empty')
             ->everyMinute()
             ->withoutOverlapping();

    // Send daily reports at 8 AM
    $schedule->job(new SendDailyReports)
             ->dailyAt('08:00')
             ->timezone('America/New_York');

    // Clear temp files weekly
    $schedule->exec('rm -rf storage/temp/*')
             ->weekly();

    // Custom maintenance tasks
    $schedule->command('maintenance:cleanup')
             ->hourly();
}
```

PROF

2. Creating Custom Commands

Generate a new command:

```
php artisan make:command DatabaseBackupCommand
```

Example Command Implementation:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Storage;
use Carbon\Carbon;

class DatabaseBackupCommand extends Command
{
    protected $signature = 'db:backup';
    protected $description = 'Create a database backup';

    public function handle()
    {
        $filename = "backup-" . Carbon::now()->format('Y-m-d-H-i-s') .
".sql";
        $command = "mysqldump --user=" . env('DB_USERNAME') .
                   " --password=" . env('DB_PASSWORD') .
                   " --host=" . env('DB_HOST') .
                   " " . env('DB_DATABASE') .
                   " > " . storage_path() . "/app/backups/" . $filename;

        $returnVar = NULL;
        $output = NULL;
        exec($command, $output, $returnVar);

        if ($returnVar === 0) {
            $this->info("Database backup created: " . $filename);

            // Upload to cloud storage
            Storage::disk('s3')->put(
                'backups/'.$filename,
file_get_contents(storage_path('app/backups/'.$filename))
            );

            return 0;
        } else {
            $this->error("Database backup failed");
            return 1;
        }
    }
}
```

PROF

3. Docker Configuration

Update your `docker-compose.yml`:

```
services:
  # ... existing services ...

  scheduler:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_scheduler
    command: >
      bash -c "echo 'Starting scheduler...' &&
      while [ true ]
      do
        php artisan schedule:run --verbose --no-interaction &
        sleep 60
      done"
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
      - redis
```

Alternative using cron in Docker:

Create a `docker/cron/laravel-cron` file:

```
* * * * * cd /var/www && php artisan schedule:run >> /dev/null 2>&1
```

Update your `docker-compose.yml`:

PROF

```
services:
  # ... existing services ...

  cron:
    image: alpine:latest
    container_name: laravel_cron
    volumes:
      - ./src:/var/www
      - ./docker/cron:/etc/crontabs/root
    command: crond -f -l 8
    depends_on:
      - app
```

4. Advanced Scheduling Techniques

Task Hooks

```
$schedule->command('reports:generate')
    ->daily()
    ->before(function () {
        // Prepare system for report generation
    })
    ->after(function () {
        // Clean up after report generation
    });
```

Maintenance Mode Awareness

```
$schedule->command('import:data')
    ->hourly()
    ->evenInMaintenanceMode();
```

Timezone Configuration

```
$schedule->command('send:daily-notifications')
    ->dailyAt('09:00')
    ->timezone('America/Chicago');
```

Job Scheduling

```
$schedule->job(new ProcessPodcasts)
    ->everyFiveMinutes()
    ->onQueue('podcasts');
```

5. Monitoring Scheduled Tasks

Logging Task Output

```
$schedule->command('inventory:update')
    ->hourly()
    ->appendOutputTo(storage_path('logs/inventory.log'));
```

Email Notifications

```
$schedule->command('reports:generate')
    ->daily()
    ->sendOutputTo(storage_path('logs/reports.log'))
    ->emailOutputTo('admin@example.com');
```

Slack Notifications

```
$schedule->command('backup:run')
    ->daily()
    ->onSuccess(function (Stringable $output) {
        Slack::to('#backups')->send("Backup succeeded!\n" . $output);
    })
    ->onFailure(function (Stringable $output) {
        Slack::to('#alerts')->send("Backup failed!\n" . $output);
    });
```

6. Complex Scheduling Examples

Business Hours Processing

```
$schedule->command('process:orders')
    ->everyFifteenMinutes()
    ->weekdays()
    ->between('8:00', '17:00');
```

Seasonal Tasks

```
$schedule->command('send:holiday-promo')
    ->daily()
    ->when(function () {
        return now()->between(
            Carbon::parse('November 15'),
            Carbon::parse('December 31')
        );
    });
```

PROF

Chained Tasks

```
$schedule->command('import:data')
    ->daily()
    ->then(function () {
        Artisan::call('process:imported-data');
    })
    ->thenPing('https://example.com/webhook/import-complete');
```

7. Testing Scheduled Tasks

Unit Testing Commands

```
public function test_backup_command()
{
    Storage::fake('s3');

    $this->artisan('db:backup')
         ->assertExitCode(0);

    Storage::disk('s3')->assertExists('backups/backup-*.sql');
}
```

Testing Schedule Definition

```
public function test_schedule_configuration()
{
    $schedule = app()->make(Schedule::class);

    $events = collect($schedule->events())
        ->filter(function ($event) {
            return stripos($event->command, 'db:backup');
        });

    $this->assertCount(1, $events);
    $this->assertEquals('0 0 * * *', $events->first()->expression);
}
```

8. Production Considerations

PROF

Supervisor Configuration

For reliable execution, use Supervisor to monitor the scheduler:

```
[program:laravel-scheduler]
process_name=%(program_name)s_%(process_num)02d
command=/bin/bash -c "php /var/www/artisan schedule:run --verbose --no-
interaction"
autostart=true
autorestart=true
user=www-data
numprocs=1
redirect_stderr=true
stdout_logfile=/var/www/storage/logs/scheduler.log
```

Locking to Prevent Overlaps

```
$schedule->command('long:process')
    ->daily()
    ->withoutOverlapping()
    ->onOneServer();
```

Error Handling

```
$schedule->command('critical:process')
    ->hourly()
    ->before(function () {
        $this->backupCurrentState();
    })
    ->onFailure(function () {
        $this->notifyAdmins();
        $this->revertToBackup();
    });
```

9. Custom Frequency Options

Extend the scheduler with custom frequencies in `AppServiceProvider`:

```
use Illuminate\Console\Scheduling\Event;

Event::macro('everyOddHour', function () {
    return $this->hourlyAt(01, 03, 05, 07, 09, 11, 13, 15, 17, 19, 21,
23);
});
```

PROF

Usage:

```
$schedule->command('sync:data')->everyOddHour();
```

10. Complete Docker Setup Example

Here's a complete Docker configuration for scheduling:

```
version: '3.8'

services:
  # ... other services ...
  scheduler:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_scheduler
    command: >
      bash -c "php artisan schedule:work"
    volumes:
      - ./src:/var/www
    environment:
      - TZ=America/New_York
    depends_on:
      - mysql
      - redis
      - app

  queue:
    build:
      context: .
      dockerfile: docker/php/Dockerfile
    container_name: laravel_queue
    command: >
      bash -c "php artisan queue:work --tries=3 --sleep=3 --timeout=120"
    volumes:
      - ./src:/var/www
    depends_on:
      - mysql
      - redis
      - app
```

This implementation provides a robust task automation system with Laravel's scheduler, complete with Docker integration, monitoring, and advanced scheduling capabilities. The system will reliably execute your scheduled tasks while providing proper logging and error handling.

PROF

Integrating Stripe Payments with Webhooks in Laravel

Here's a comprehensive guide to integrate Stripe payments for subscriptions and one-time charges with webhook support in your Laravel application.

1. Install Required Packages

```
composer require stripe/stripe-php laravel/cashier
```

2. Set Up Configuration

Add to your `.env`:

```
STRIPE_KEY=your_stripe_publishable_key
STRIPE_SECRET=your_stripe_secret_key
STRIPE_WEBHOOK_SECRET=your_stripe_webhook_secret
CASHIER_CURRENCY=usd
CASHIER_CURRENCY_LOCALE=en
```

Publish Cashier migrations:

```
php artisan vendor:publish --tag="cashier-migrations"
php artisan migrate
```

3. Prepare Your User Model

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

4. Set Up Routes

PROF

```
// routes/web.php
Route::post('/stripe/webhook',
'\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook');

// One-time charge
Route::post('/charge', [PaymentController::class, 'charge'])-
>name('charge');

// Subscription
Route::post('/subscribe', [SubscriptionController::class, 'subscribe'])-
>name('subscribe');
Route::post('/cancel-subscription', [SubscriptionController::class,
'cancel'])->name('cancel-subscription');
```

5. Create Controllers

For one-time charges:

```
// app/Http/Controllers/PaymentController.php

use Illuminate\Http\Request;
use Stripe\Stripe;
use Stripe\PaymentIntent;

class PaymentController extends Controller
{
    public function charge(Request $request)
    {
        Stripe::setApiKey(config('services.stripe.secret'));

        try {
            $paymentIntent = PaymentIntent::create([
                'amount' => $request->amount * 100, // in cents
                'currency' => config('cashier.currency'),
                'payment_method' => $request->payment_method,
                'confirm' => true,
                'description' => $request->description,
                'metadata' => [
                    'user_id' => auth()->id(),
                    'product_id' => $request->product_id
                ],
            ]);

            // Save the payment to your database
            auth()->user()->payments()->create([
                'stripe_id' => $paymentIntent->id,
                'amount' => $paymentIntent->amount / 100,
                'status' => $paymentIntent->status,
            ]);

            return response()->json(['success' => true]);

        } catch (\Exception $e) {
            return response()->json(['error' => $e->getMessage()], 500);
        }
    }
}
```

PROF

For subscriptions:

```
// app/Http/Controllers/SubscriptionController.php

use Illuminate\Http\Request;

class SubscriptionController extends Controller
```

```
{
    public function subscribe(Request $request)
    {
        $user = $request->user();
        $paymentMethod = $request->payment_method;
        $planId = $request->plan_id;

        try {
            $user->createOrGetStripeCustomer();
            $user->updateDefaultPaymentMethod($paymentMethod);

            $subscription = $user->newSubscription('default', $planId)
                ->create($paymentMethod, [
                    'email' => $user->email,
                ]);

            return response()->json(['success' => true]);

        } catch (\Exception $e) {
            return response()->json(['error' => $e->getMessage()], 500);
        }
    }

    public function cancel(Request $request)
    {
        $request->user()->subscription('default')->cancel();

        return response()->json(['success' => true]);
    }
}
```

6. Set Up Webhooks

Create a webhook handler:

PROF

```
// app/Http/Controllers/WebhookController.php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as
CashierController;

class WebhookController extends CashierController
{
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle successful invoice payment (for subscriptions)
        $user = $this->getUserByStripeId($payload['data']['object']
['customer']);
```

```
        // Your logic here (e.g., send email, update database)

        return response('Webhook Handled', 200);
    }

    public function handleChargeSucceeded($payload)
    {
        // Handle successful one-time charge
        $charge = $payload['data']['object'];

        // Your logic here

        return response('Webhook Handled', 200);
    }

    public function handleCustomerSubscriptionDeleted($payload)
    {
        // Handle subscription cancellation
        $user = $this->getUserByStripeId($payload['data']['object']
['customer']);

        // Your logic here

        return response('Webhook Handled', 200);
    }

    // Add more webhook handlers as needed
}
```

7. Frontend Implementation (JavaScript)

PROF

```
// resources/js/stripe.js

const stripe = Stripe(process.env.MIX_STRIPE_KEY);

// For one-time payment
async function handlePayment(amount, description) {
    const { error, paymentIntent } = await
stripe.confirmCardPayment(clientSecret, {
        payment_method: {
            card: elements.getElement('card'),
            billing_details: {
                name: document.getElementById('card-holder-name').value,
            },
        }
    });

    if (error) {
        console.error(error);
    } else {
        await fetch('/charge', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'X-CSRF-TOKEN': document.querySelector('meta[name="csrf-
token"]').content
            },
            body: JSON.stringify({
                payment_method: paymentIntent.payment_method,
                amount: amount,
                description: description
            })
        });
    }
}

// For subscription
async function handleSubscribe(planId) {
    const { error, paymentMethod } = await stripe.createPaymentMethod(
        'card', elements.getElement('card'), {
            billing_details: {
                name: document.getElementById('card-holder-name').value,
            },
        }
    );

    if (error) {
        console.error(error);
    } else {
        await fetch('/subscribe', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'X-CSRF-TOKEN': document.querySelector('meta[name="csrf-
token"]').content
            },
            body: JSON.stringify({
                payment_method: paymentMethod.id,
                plan_id: planId
            })
        });
    }
}
```

PROF

8. Testing Webhooks Locally

Use the Stripe CLI to test webhooks locally:

```
stripe listen --forward-to localhost:8000/stripe/webhook
```

9. Important Webhook Events to Handle

Here are key Stripe events you should handle:

- `invoice.payment_succeeded` - Subscription payment succeeded
- `invoice.payment_failed` - Subscription payment failed
- `charge.succeeded` - One-time payment succeeded
- `charge.failed` - One-time payment failed
- `customer.subscription.deleted` - Subscription cancelled/ended
- `customer.subscription.updated` - Subscription changed

10. Security Considerations

1. Always verify webhook signatures:

```php
$payload = @file_get_contents('php://input');
$sig_header = $_SERVER['HTTP_STRIPE_SIGNATURE'];
$event = null;

try {
    $event = \Stripe\Webhook::constructEvent(
        $payload, $sig_header, config('services.stripe.webhook_secret')
    );
} catch(\UnexpectedValueException $e) {
    // Invalid payload
    http_response_code(400);
    exit();
} catch(\Stripe\Exception\SignatureVerificationException $e) {
    // Invalid signature
    http_response_code(400);
    exit();
}
```

PROF

2. Use idempotency keys for critical operations to prevent duplicate processing.

3. Implement proper error handling and logging for all webhook events.

This implementation provides a solid foundation for both subscription and one-time payments with proper webhook handling in your Laravel application.

Setting Up CI/CD with GitHub Actions for Laravel Deployment

Here's a comprehensive guide to set up automated deployment for your Laravel application using GitHub Actions.

1. Basic GitHub Actions Workflow

Create a `.github/workflows/laravel.yml` file in your repository:

```
name: Laravel CI/CD

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  tests:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:5.7
        env:
          MYSQL_ROOT_PASSWORD: secret
          MYSQL_DATABASE: laravel
        ports:
          - 3306:3306
        options: --health-cmd="mysqladmin ping" --health-interval=10s --
health-timeout=5s --health-retries=3

    steps:
    - uses: actions/checkout@v4

    - name: Setup PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.2'
        extensions: mbstring, ctype, fileinfo, openssl, PDO, mysql,
pgsql, sqlite, gd, exif, pcntl, bcmath, intl
        coverage: none

    - name: Install dependencies
      run: |
        composer install --prefer-dist --no-interaction --no-progress

    - name: Copy .env
      run: |
        cp .env.example .env
        php artisan key:generate

    - name: Directory Permissions
      run: |
        mkdir -p storage/framework/{sessions,views,cache}
        chmod -R 777 storage bootstrap/cache
```

PROF

```
    - name: Execute tests
      env:
        DB_CONNECTION: mysql
        DB_DATABASE: laravel
        DB_USERNAME: root
        DB_PASSWORD: secret
      run: |
        php artisan migrate:fresh --env=testing --force
        php artisan test
```

2. Deployment Workflow (to a Server)

For deployment to a server (like a VPS), add this to your workflow:

```
  deploy:
    needs: tests
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4

    - name: Install dependencies
      run: composer install --prefer-dist --no-dev --no-interaction --
no-progress

    - name: Upload files via SSH
      uses: appleboy/scp-action@v0.1.3
      with:
        host: ${{ secrets.SSH_HOST }}
        username: ${{ secrets.SSH_USERNAME }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        port: ${{ secrets.SSH_PORT }}
        source: "./"
        target: "/var/www/your-project"

    - name: Run deployment commands
      uses: appleboy/ssh-action@v0.1.10
      with:
        host: ${{ secrets.SSH_HOST }}
        username: ${{ secrets.SSH_USERNAME }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        port: ${{ secrets.SSH_PORT }}
        script: |
          cd /var/www/your-project
          composer install --no-dev
          php artisan migrate --force
          php artisan optimize:clear
          php artisan optimize
```

PROF

3. Required GitHub Secrets

You'll need to set up these secrets in your GitHub repository (Settings > Secrets):

- `SSH_HOST`: Your server IP or domain
- `SSH_USERNAME`: SSH username (usually "root" or your sudo user)
- `SSH_PRIVATE_KEY`: Your private SSH key
- `SSH_PORT`: SSH port (usually 22)

4. Alternative: Deployment to Shared Hosting

For shared hosting without SSH access:

```
  deploy:
    needs: tests
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4

    - name: Install dependencies
      run: composer install --prefer-dist --no-dev --no-interaction --
no-progress

    - name: Zip artifacts
      run: zip -r deploy.zip . -x '*.git*'

    - name: Upload artifact
      uses: actions/upload-artifact@v3
      with:
        name: deployment-package
        path: deploy.zip
```

PROF

Then manually download the artifact and upload to your hosting.

5. Additional Optimizations

Consider adding these steps to your workflow:

```
- name: Cache Composer packages
  uses: actions/cache@v3
  with:
    path: vendor
    key: ${{ runner.os }}-php-${{ hashFiles('**/composer.lock') }}
    restore-keys: |
      ${{ runner.os }}-php-

- name: Cache NPM packages
  if: steps.yarn-cache.outputs.cache-hit != 'true'
```

```
    uses: actions/cache@v3
    with:
      path: node_modules
      key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
      restore-keys: |
        ${{ runner.os }}-node-

  - name: Install NPM dependencies
    run: npm ci && npm run prod
```

6. Environment-Specific Workflows

For different environments (staging/production):

```
on:
  push:
    branches:
      - main
    paths-ignore:
      - 'docs/**'
      - 'README.md'

  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy to'
        required: true
        default: 'staging'
        type: choice
        options:
        - staging
        - production
```

PROF

This setup provides:

1. Automated testing on every push/pull request
2. Secure deployment to your server
3. Proper environment handling
4. Caching for faster builds
5. Flexibility for different deployment targets

Remember to adjust paths, server details, and commands according to your specific project requirements.