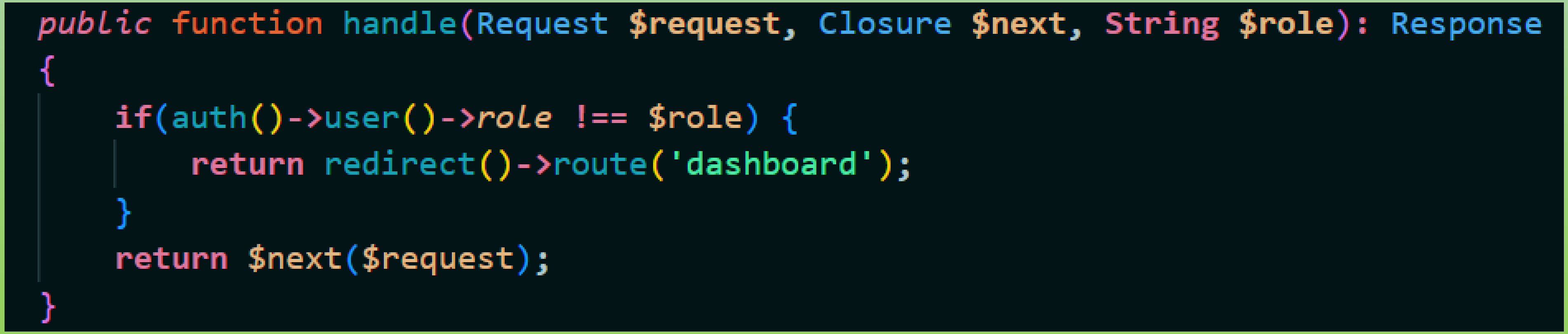
Advanced Laravel Experience

By Aung Khant Kyaw

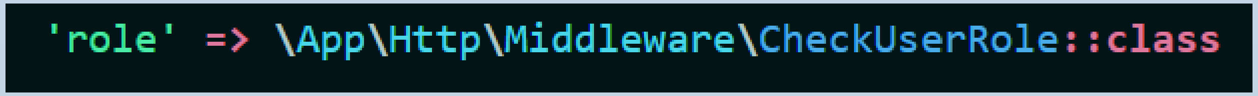
Writing custom middleware

- middleware acts as a bridge between HTTP requests and application's routes
- can be used to perform various tasks, such as authentication, logging, and request filtering, before a request reaches the intended controller.
- CheckUserRole custom middleware example >>>>>



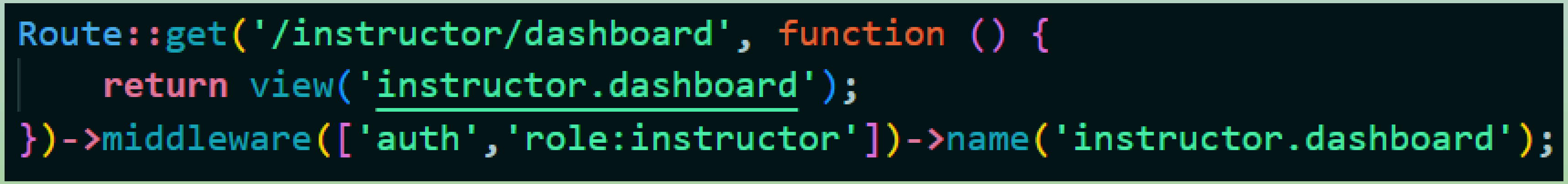
```php
public function handle(Request $request, Closure $next, String $role): Response
{
    if(auth()->user()->role !== $role) {
        return redirect()->route('dashboard');
    }
    return $next($request);
}
```

- **Step 2: Register the middleware in app/Http/kernel.php**

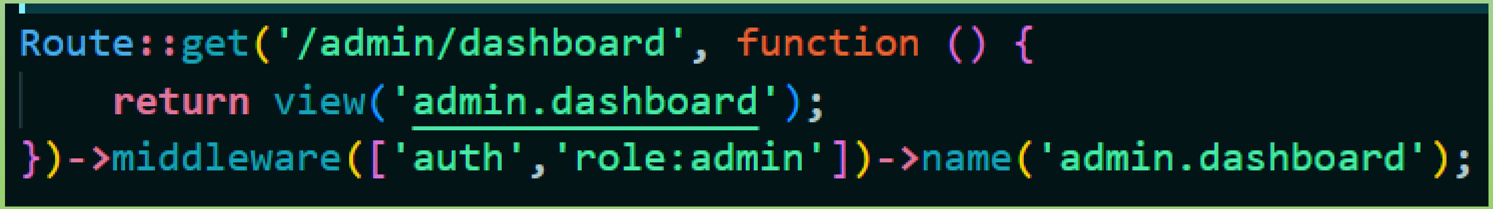


```
'role' => \App\Http\Middleware\CheckUserRole::class
```

- **Step 3: Apply Middleware to Routes**



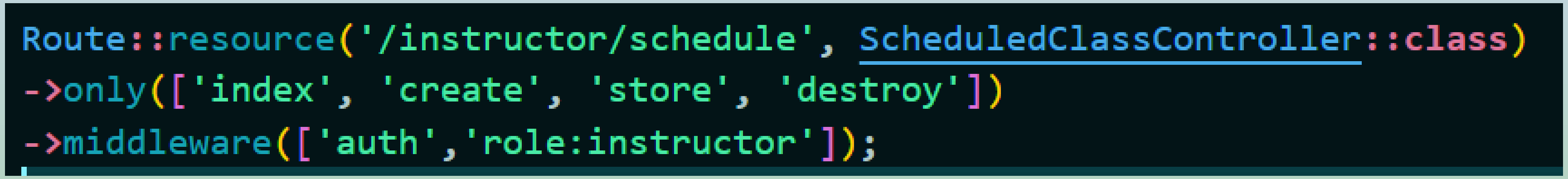
```
Route::get('/instructor/dashboard', function () {
    return view('instructor.dashboard');
})->middleware(['auth','role:instructor'])->name('instructor.dashboard');
```



```
Route::get('/admin/dashboard', function () {
    return view('admin.dashboard');
})->middleware(['auth','role:admin'])->name('admin.dashboard');
```

How a resource controller is different from a normal controller

- includes methods for standard CRUD operations with predefined naming conventions.

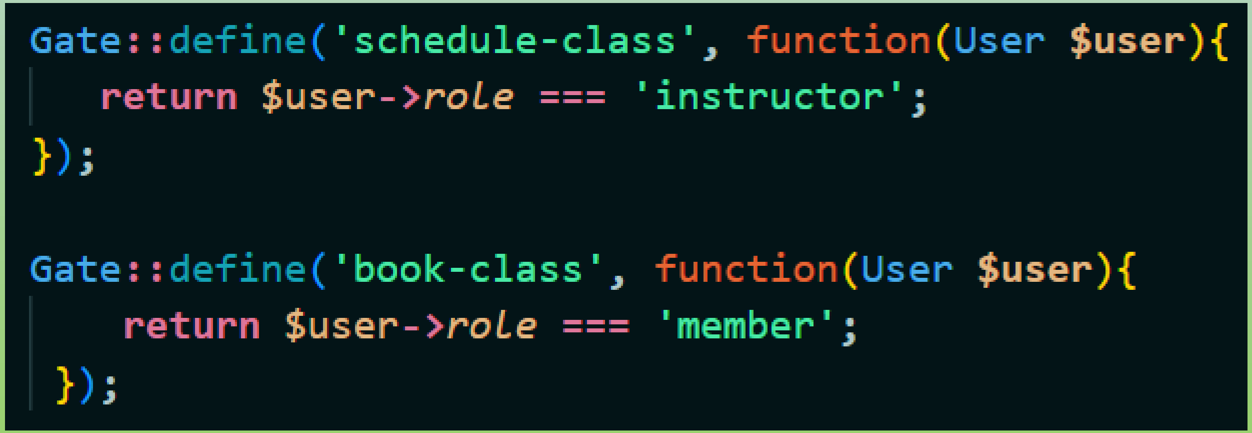


```
Route::resource('/instructor/schedule', ScheduledClassController::class)
->only(['index', 'create', 'store', 'destroy'])
->middleware(['auth','role:instructor']);
```

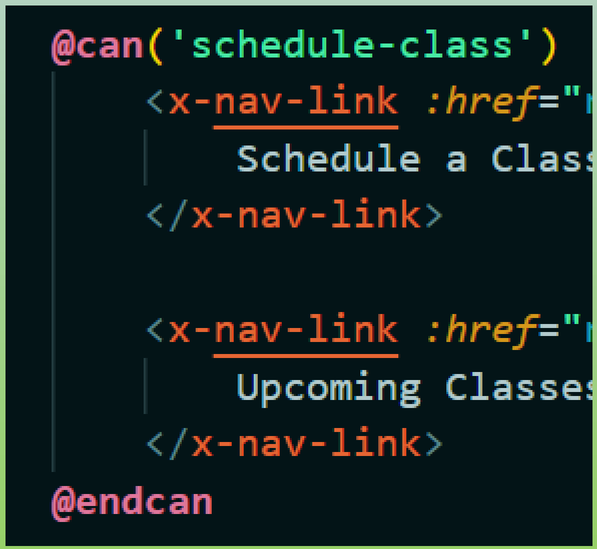
- Consistent Routing
- Reduce Boilerplate code (repetitive code)
  (E.g HTML, CSS Structure/ constructors and getters,setters

Gates and Policies

- used for implementing authorization and access control in your application.
- Define gates in ‘AuthServiceProvider’ class using Gate facade.

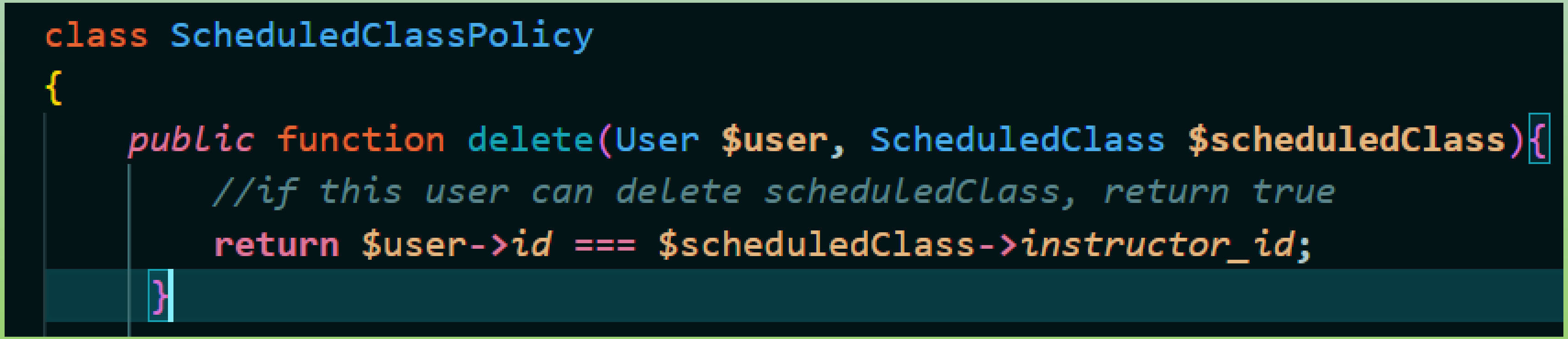


```
Gate::define('schedule-class', function(User $user){
    return $user->role === 'instructor';
});

Gate::define('book-class', function(User $user){
     return $user->role === 'member';
 });
```



```
@can('schedule-class')
    <x-nav-link :href="
        Schedule a Clas
    </x-nav-link>

    <x-nav-link :href="
        Upcoming Classe
    </x-nav-link>
@endcan
```

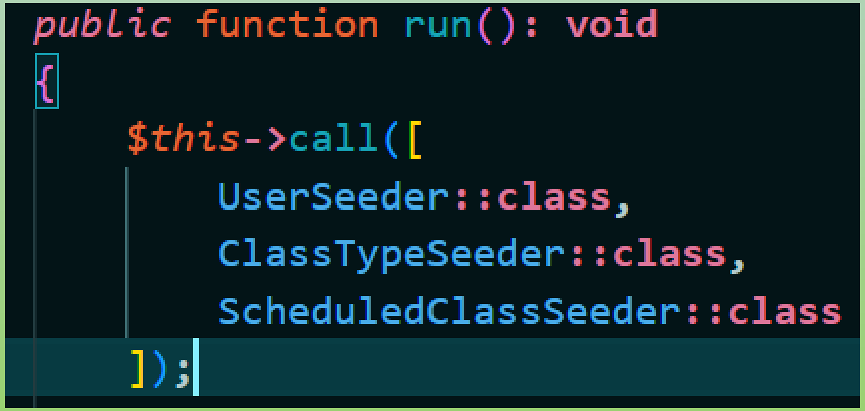
- Policies are used to group authorization logic for a specific model or resource.
- The naming convention for policies is based on the model. (E.g 'PostPolicy' for the 'Post' model)
- If you don't follow naming convention, manually register the policy in 'AuthServiceProvider'



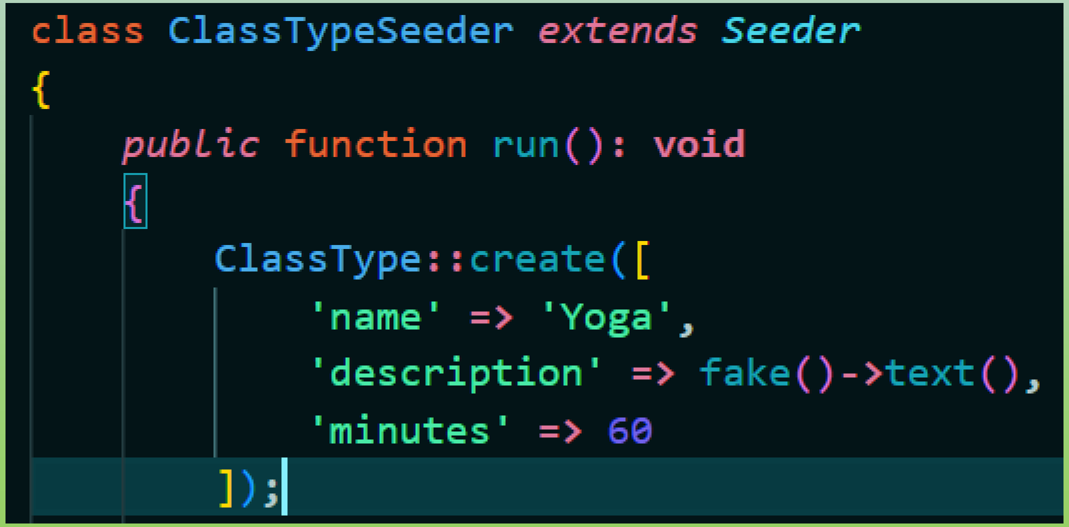
```
class ScheduledClassPolicy
{
    public function delete(User $user, ScheduledClass $scheduledClass){
        //if this user can delete scheduledClass, return true
        return $user->id === $scheduledClass->instructor_id;
    }
```

Seeding and Factories

- Seeding refers to the process of populating your application's database tables with predefined data.
- In 'DatabaseSeeder' class, we can specify which seeders should run and their order by using the 'call' method.

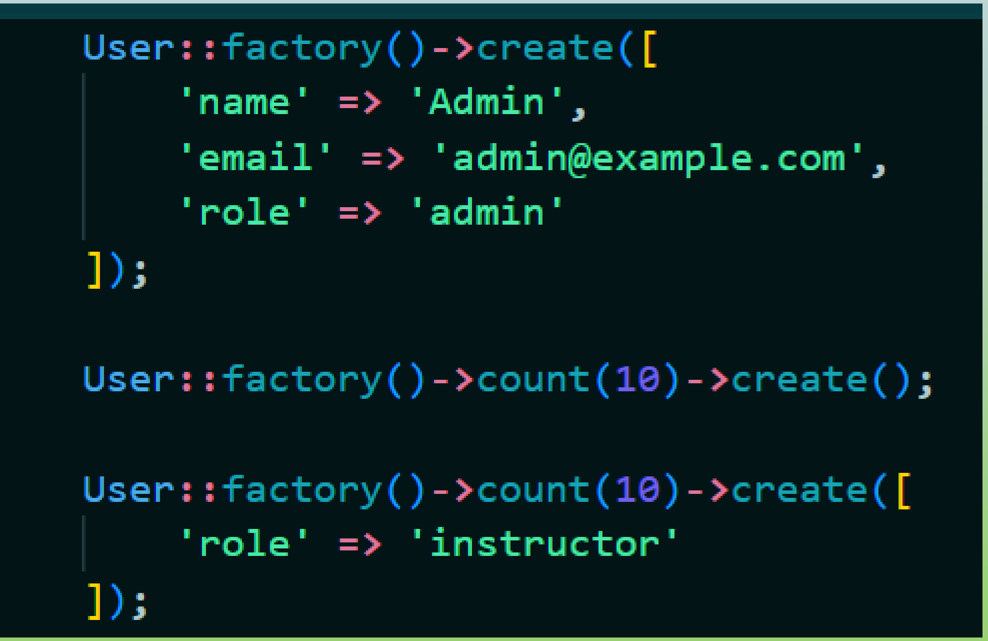


```php
public function run(): void
{
    $this->call([
        UserSeeder::class,
        ClassTypeSeeder::class,
        ScheduledClassSeeder::class
    ]);
```

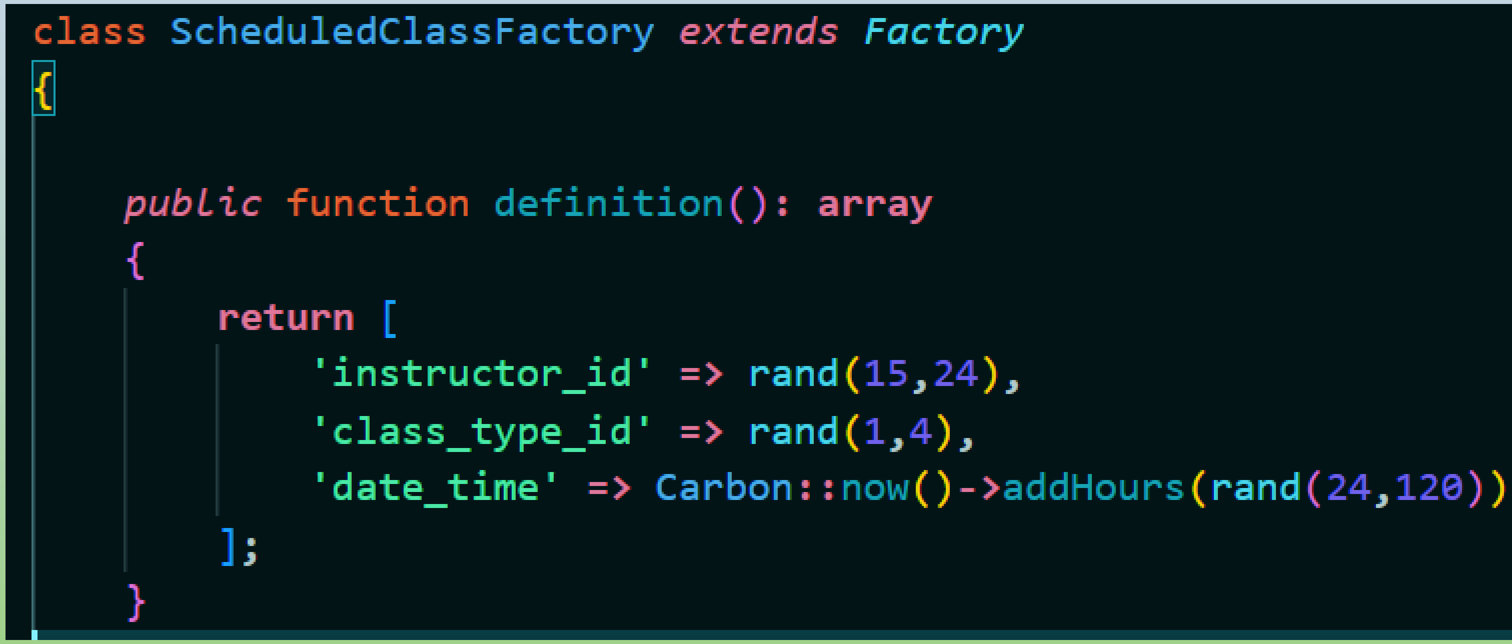


```php
class ClassTypeSeeder extends Seeder
{
    public function run(): void
    {
        ClassType::create([
            'name' => 'Yoga',
            'description' => fake()->text(),
            'minutes' => 60
        ]);
```

- Factories provide a way to generate random or predefined data for testing and database seeding.

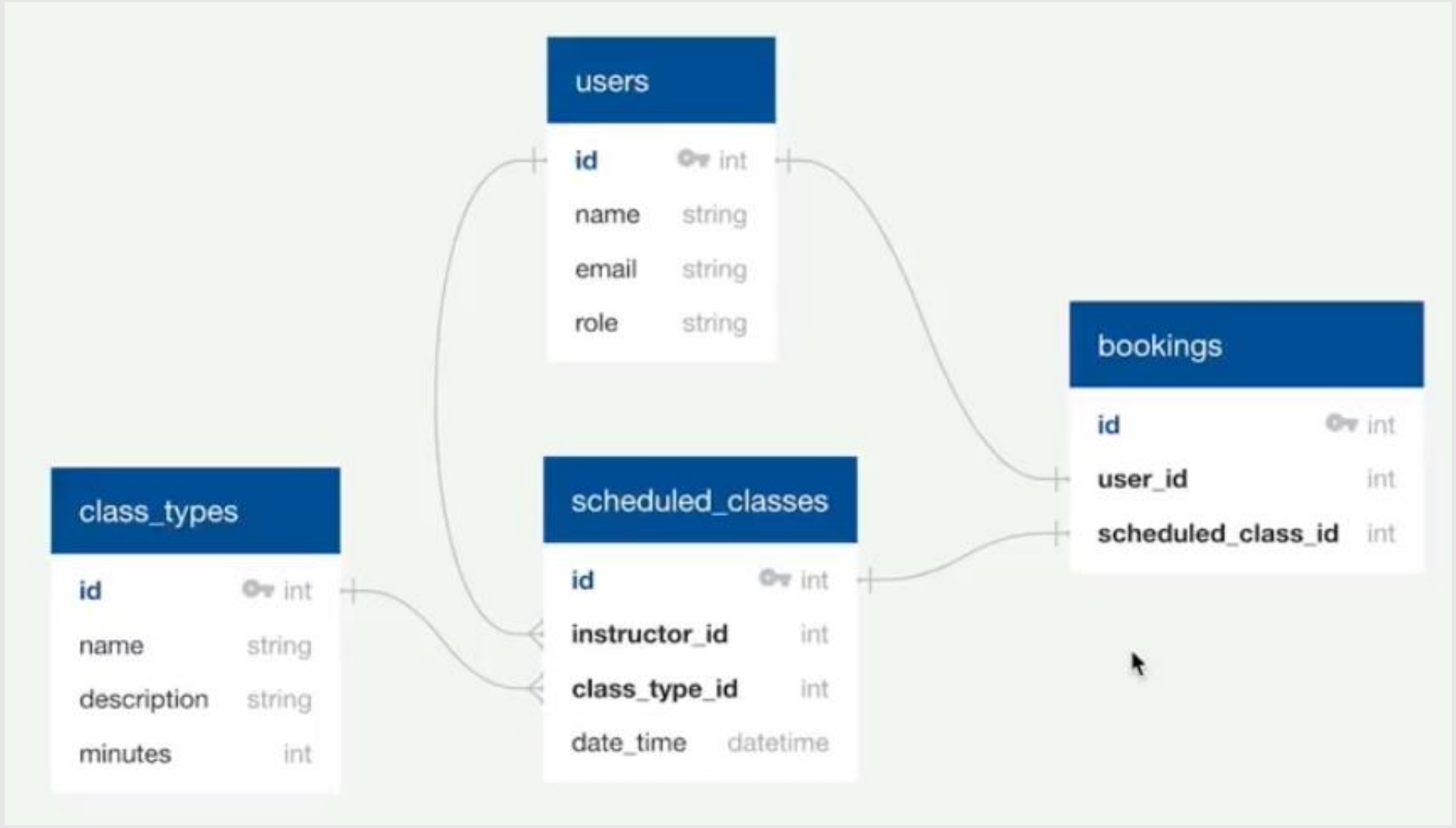


```php
User::factory()->create([
    'name' => 'Admin',
    'email' => 'admin@example.com',
    'role' => 'admin'
]);

User::factory()->count(10)->create();

User::factory()->count(10)->create([
    'role' => 'instructor'
]);
```



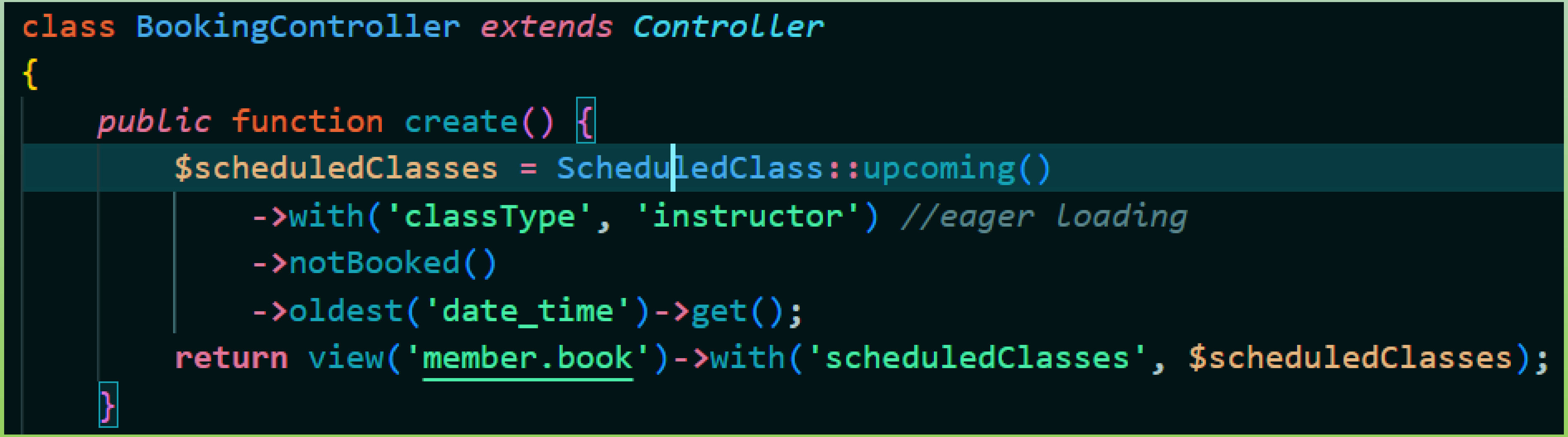
```php
class ScheduledClassFactory extends Factory
{

    public function definition(): array
    {
        return [
            'instructor_id' => rand(15,24),
            'class_type_id' => rand(1,4),
            'date_time' => Carbon::now()->addHours(rand(24,120))
        ];
    }
```

Many-to-Many Relationships



Eager loading

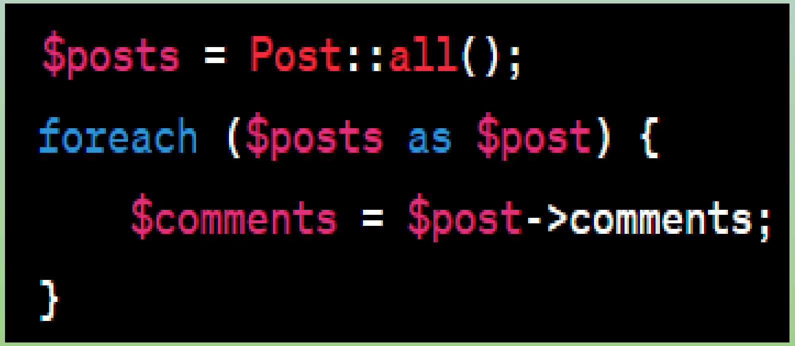
- Eager loading is a technique used to load related data along with the main model in a single query, rather than making additional queries for each related item individually.
- In Laravel, we can use eager loading with the ‘with’ method.



```php
class BookingController extends Controller
{
    public function create() {
        $scheduledClasses = ScheduledClass::upcoming()
            ->with('classType', 'instructor') //eager loading
            ->notBooked()
            ->oldest('date_time')->get();
        return view('member.book')->with('scheduledClasses', $scheduledClasses);
    }
```

Lazy Loading

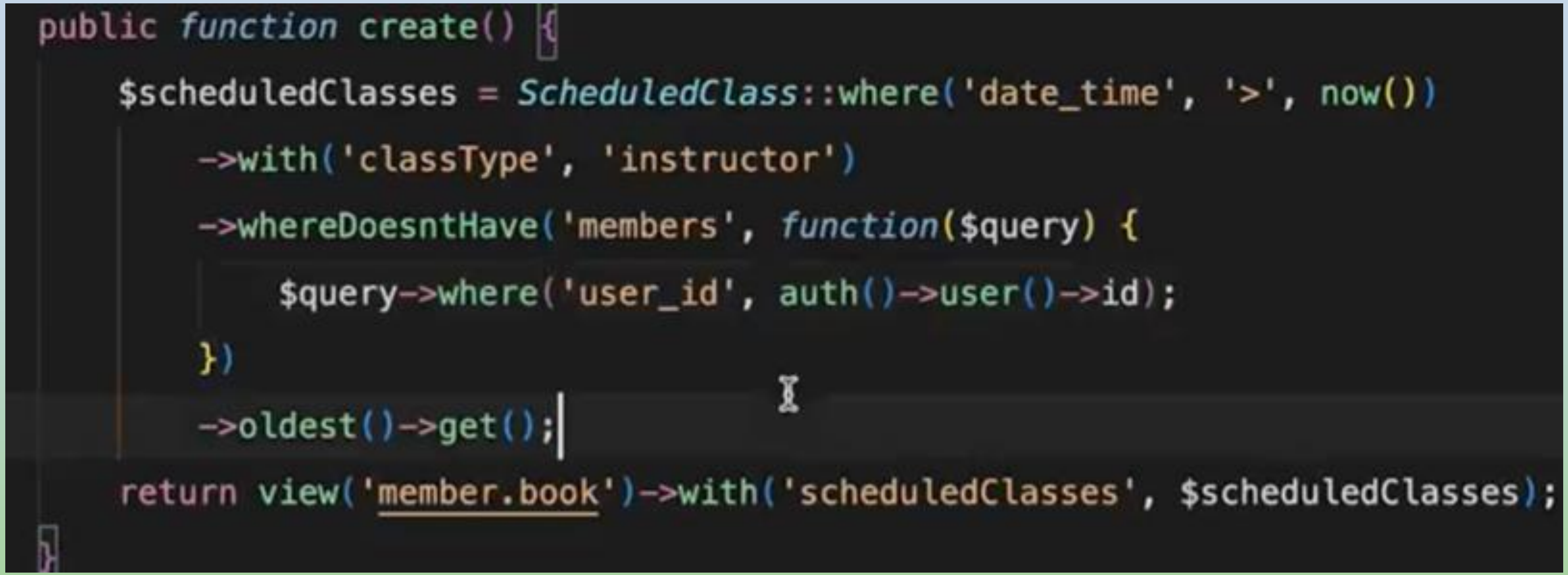
- Lazy loading is the default behavior in Laravel.
- Lazy loading can lead to the N+1 query problem, where you make N+1 queries when retrieving N main records and their related records.



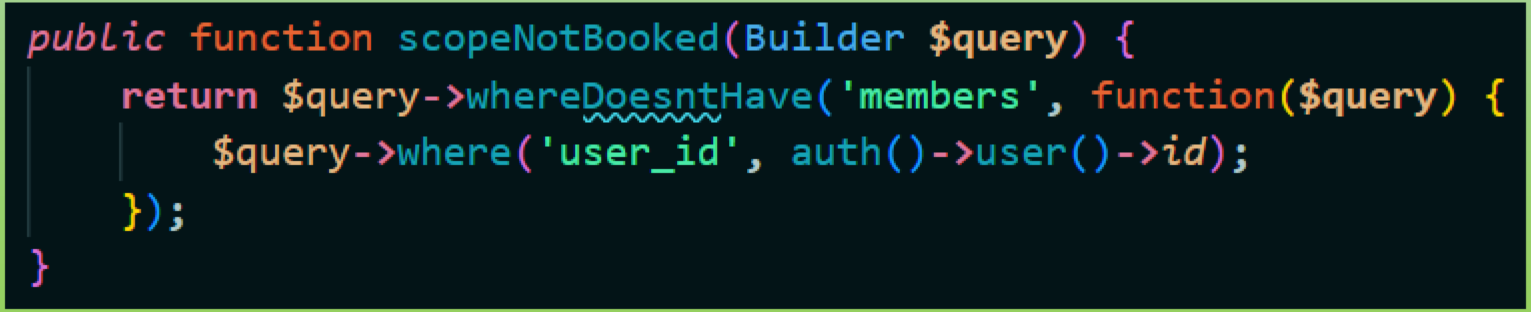
```php
$posts = Post::all();
foreach ($posts as $post) {
    $comments = $post->comments;
}
```

- In this example, one query to retrieve all posts and then an additional query for each post to retrieve its comments.

Complex queries and Query scopes



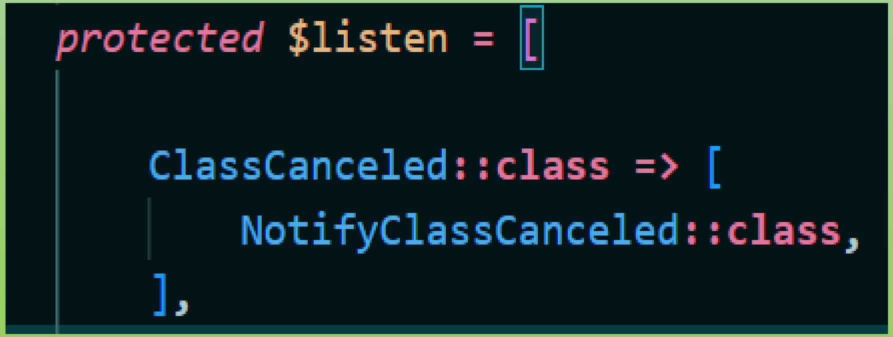
```php
public function create() {
    $scheduledClasses = ScheduledClass::where('date_time', '>', now())
        ->with('classType', 'instructor')
        ->whereDoesntHave('members', function($query) {
            $query->where('user_id', auth()->user()->id);
        })
        ->oldest()->get();
    return view('member.book')->with('scheduledClasses', $scheduledClasses);
}
```



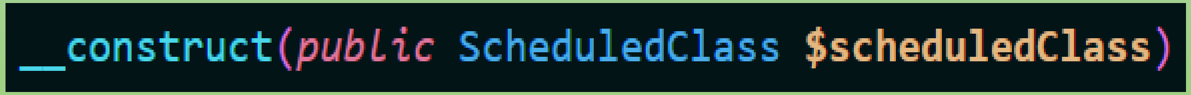
```php
public function scopeNotBooked(Builder $query) {
    return $query->whereDoesntHave('members', function($query) {
        $query->where('user_id', auth()->user()->id);
    });
}
```

Events and Listeners

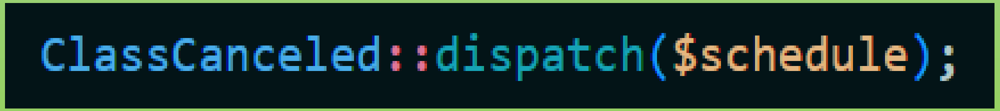
- An event is an occurrence or action that happens within application, such as a user registering, a new comment being posted, a purchase being made, and more.
- A listener is a class that handles a specific event when it's fired.
- Listeners are responsible for performing actions or logic in response to the event.
- Register the event and listener in 'EventServiceProvider'



```
protected $listen = [

    ClassCanceled::class => [
        NotifyClassCanceled::class,
    ],
```

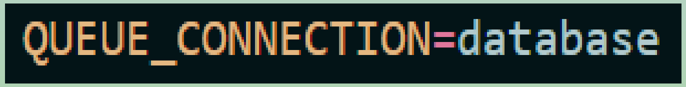


```
__construct(public ScheduledClass $scheduledClass)
```

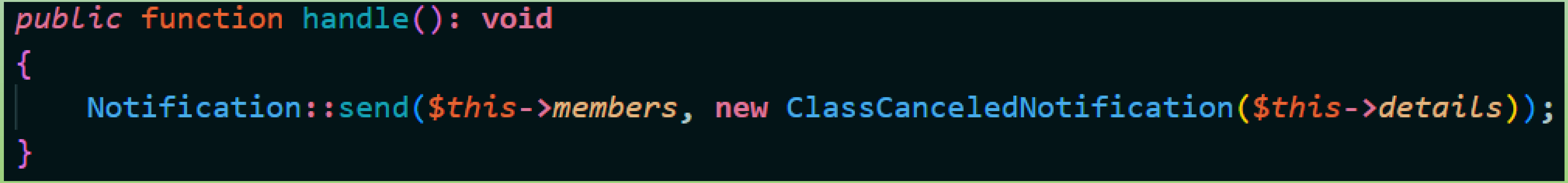


```
ClassCanceled::dispatch($schedule);
```

Jobs and Queues

- A job is a unit of work that represents a task you want to perform.
- A queue is a system for managing jobs that need to be executed.
- Sending emails, processing images, or performing calculations in frontend is too time-consuming.
- To configure in .env file >>> 

```
QUEUE_CONNECTION=database
```

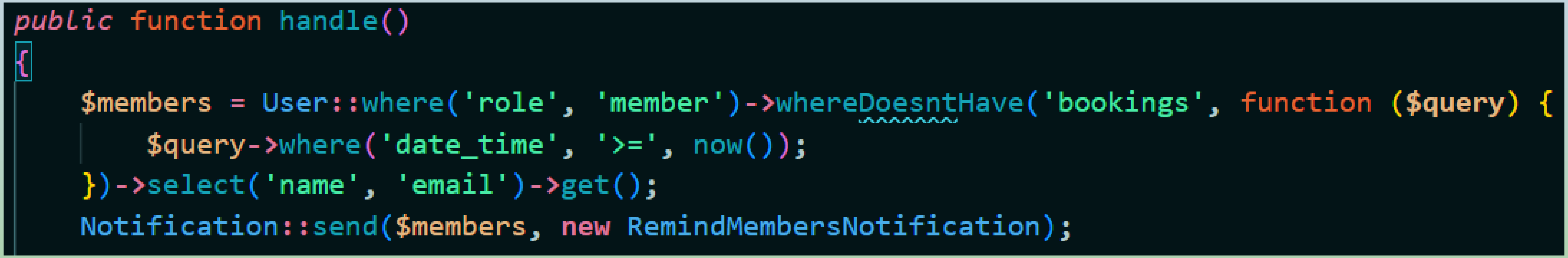


```php
public function handle(): void
{
    Notification::send($this->members, new ClassCanceledNotification($this->details));
}
```

- Run queue worker>> php artisan queue:work

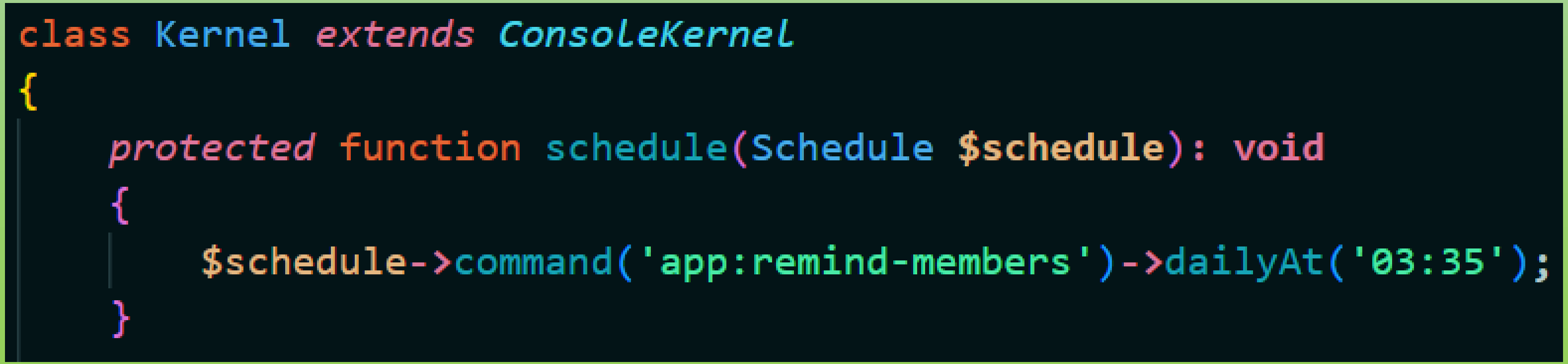
Schedule Tasks Automatically

- First, create the Artisan commands that you want to schedule.
- Second, write logic in command to do that task.



```php
public function handle()
{
    $members = User::where('role', 'member')->whereDoesntHave('bookings', function ($query) {
        $query->where('date_time', '>=', now());
    })->select('name', 'email')->get();
    Notification::send($members, new RemindMembersNotification);
```

- Third, define scheduled tasks in app/Console/Kernal.php.



```php
class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule): void
    {
        $schedule->command('app:remind-members')->dailyAt('03:35');
    }
```

Thank you for listening