Material de Estudio: Desarrollo Backend I

Introducción

Esta es una guía de estudio sobre el desarrollo de aplicaciones web y APIs con Laravel. Este documento consolida una serie de tutoriales prácticos en un manual coherente, diseñado para llevar a los estudiantes desde los conceptos fundamentales de la base de datos hasta la creación de una API REST segura y funcional.

El objetivo es ayudarte a recordar y a resumir todo el proceso llevado a cabo a lo largo de la materia, desde estructurar tu base de datos con migraciones, a interactuar con ella de forma elegante usando el ORM Eloquent, a poblarla con datos realistas mediante Seeders y Factories, y a construir la lógica de negocio con Rutas y Controladores. Finalmente, expondremos esta lógica como una API REST y la aseguraremos con un sistema de autenticación y autorización profesional.

Índice del Documento

Módulo 1: Fundamentos de la Base de Datos con Eloquent

1.1. Introducción a Migraciones y Modelos (ORM Eloquent)

En Laravel, la interacción con la base de datos se realiza principalmente a través de dos componentes:

- **Migraciones**: Son como un sistema de control de versiones para tu base de datos. Permiten definir la estructura de las tablas (el "esquema") en archivos PHP, facilitando su modificación y compartición entre desarrolladores.
- **Modelos Eloquent**: Son clases de PHP que representan una tabla de la base de datos. Eloquent es el **ORM** (Object-Relational Mapper) de Laravel, lo que significa que nos permite trabajar con nuestras tablas como si fueran objetos, simplificando enormemente operaciones como inserciones, búsquedas y actualizaciones.

1.2. Creación de Migraciones: El Control de Versiones de tu BD

Para crear una migración, usamos un comando de Artisan:

```
php artisan make:migration create_products_table```

Este comando crea un archivo en `database/migrations/`. Cada archivo de migración contiene dos m
*   `up()`: Se ejecuta para aplicar los cambios en la base de datos (crear tablas, añadir column
*   `down()`: Se ejecuta para revertir los cambios del método `up()`.

**Ejemplo de Migración:**

```php
// database/migrations/YYYY_MM_DD_HHMMSS_create_products_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateProductsTable extends Migration
{
    public function up()
    {
        Schema::create('products', function (Blueprint $table) {
            $table->id(); // BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY
            $table->string('name'); // VARCHAR(255)
            $table->text('description')->nullable(); // TEXT, puede ser nulo
            $table->decimal('price', 8, 2); // DECIMAL con 8 dígitos totales y 2 decimales
            $table->integer('stock'); // INTEGER
            $table->boolean('is_active')->default(true); // BOOLEAN con valor por defecto
            $table->timestamps(); // Crea las columnas created_at y updated_at
        });
    }

    public function down()
    {
        Schema::dropIfExists('products'); // Elimina la tabla si existe
    }
}
```

Para ejecutar todas las migraciones pendientes:

```
php artisan migrate
```
```

1.3. Tipos de Datos y Modificadores Comunes

Método de Schema	Tipo SQL Equivalente	Descripción
`$table->id();`	`BIGINT AUTO_INCREMENT PK`	Clave primaria autoincremental.
`$table->string('name');`	`VARCHAR(255)`	Cadena de texto de longitud variable.
`$table->text('desc');`	`TEXT`	Texto largo.
`$table->integer('qty');`	`INT`	Número entero.
`$table->boolean('active');`	`BOOLEAN` / `TINYINT(1)`	Valor booleano.
`$table->foreignId('user_id');`	`BIGINT UNSIGNED`	Clave foránea.
`->nullable()`		Permite que la columna acepte valores `NULL`.
`->default($value)`		Define un valor por defecto.
`->unique()`		Crea un índice único para la columna.
`->constrained()`		Añade la restricción de clave foránea.

1.4. Modelos Eloquent y sus Convenciones

Para crear un modelo, usamos Artisan:

```
php artisan make:model Product
```

Laravel asume convenciones para conectar el modelo con la tabla:

- **Nombre de la clase:** `Product` (singular, PascalCase)
- **Nombre de la tabla:** `products` (plural, snake_case)

Ejemplo de Modelo Básico:

```php
// app/Models/Product.php
namespace App\Models;

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

class Product extends Model
{
    use HasFactory;

    // Campos que se pueden asignar masivamente (protección contra Mass Assignment)
    protected $fillable = [
        'name',
        'description',
        'price',
        'stock',
        'is_active',
    ];

    // Define cómo ciertos atributos deben ser convertidos a tipos de datos
    protected $casts = [
        'is_active' => 'boolean',
        'price' => 'decimal:2',
    ];
}
```

1.5. El Poder de los Enums para un Código Robusto

Los **Enums** (enumeraciones) de PHP 8.1+ son una forma excelente de definir un conjunto fijo de valores posibles para un campo, evitando errores por datos incorrectos.

Ejemplo de Enum:

```php
// app/Enums/PostStatus.php
namespace App\Enums;

enum PostStatus: string
{
    case DRAFT = 'draft';
    case PUBLISHED = 'published';
    case ARCHIVED = 'archived';

    // Método para obtener todos los valores como un array
    public static function values(): array
    {
        return array_column(self::cases(), 'value');
    }

    // Método para obtener una etiqueta legible para la UI
    public function label(): string
    {
        return match($this) {
            self::DRAFT => 'Borrador',
            self::PUBLISHED => 'Publicado',
            self::ARCHIVED => 'Archivado',
        };
    }
}
```

Uso en una Migración:

```php
use App\Enums\PostStatus;
// ...
$table->enum('status', PostStatus::values());
```

Uso en un Modelo:

```php
// app/Models/Post.php
protected $casts = [
    'status' => PostStatus::class, // Conversión automática
];
```

Módulo 2: Poblando la Base de Datos (Seeders y Factories)

Ahora que la estructura está definida, necesitamos datos para trabajar.

2.1. ¿Qué son los Seeders?

Los **Seeders** son clases diseñadas para insertar datos iniciales o de prueba en la base de datos. Son fundamentales para tener un entorno de desarrollo consistente.

Para crear un seeder:

```
php artisan make:seeder ChannelSeeder
```

2.2. Creación de un Seeder Básico: `create`, `firstOrCreate` y `updateOrCreate`

Dentro del método `run()` del seeder, podemos usar varios métodos de Eloquent:

- `create()` : Siempre crea un nuevo registro. Fallará si hay restricciones de unicidad.
- `firstOrCreate()` : **Recomendado**. Busca un registro con los atributos del primer array; si no lo encuentra, lo crea con los atributos de ambos arrays.
- `updateOrCreate()` : Busca un registro; si lo encuentra, lo actualiza; si no, lo crea.

Ejemplo con `firstOrCreate`:

```php
// database/seeders/ChannelSeeder.php
use App\Models\Channel;

public function run(): void
{
    $channels = [
        ['name' => 'Departamento de Comunicación', 'type' => 'department'],
        ['name' => 'Instituto de Investigación', 'type' => 'institute'],
    ];

    foreach ($channels as $channelData) {
        Channel::firstOrCreate(
            ['name' => $channelData['name']], // Busca por este campo único
            $channelData // Si no existe, crea con todos estos datos
        );
    }

    $this->command->info('Canales creados exitosamente!');
}
```

2.3. ¿Qué son las Factories y Faker?

Cuando necesitamos una gran cantidad de datos de prueba realistas (ej. 100 usuarios, 500 productos), escribirlos a mano es inviable. Aquí es donde entran las **Factories**.

Las Factories definen una "receta" para crear modelos, usando la librería **Faker** para generar datos falsos pero con apariencia real (nombres, correos, párrafos, etc.).

Para crear una factory:

```
php artisan make:factory ProductFactory
```

2.4. Creación de una Factory para Datos Realistas

```php
// database/factories/ProductFactory.php
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ProductFactory extends Factory
{
    public function definition(): array
    {
        return [
            'name' => $this->faker->words(3, true), // 3 palabras como texto
            'description' => $this->faker->paragraph(), // Un párrafo
            'price' => $this->faker->randomFloat(2, 10, 500), // decimal con 2 dígitos, entre 1
            'stock' => $this->faker->numberBetween(0, 100), // entero entre 0 y 100
            'is_active' => $this->faker->boolean(80), // 80% de probabilidad de ser true
        ];
    }
}
```

2.5. El Seeder Orquestador: `DatabaseSeeder.php`

Este es el archivo principal que se ejecuta con `php artisan db:seed`. Desde aquí, llamamos a otros seeders en el orden correcto para respetar las dependencias.

```php
// database/seeders/DatabaseSeeder.php
public function run(): void
{
    $this->command->info('Iniciando el sembrado de la base de datos...');

    $this->call([
        ChannelSeeder::class,   // Debe ejecutarse antes de los que dependen de Channel
        UserSeeder::class,      // Debe ejecutarse antes de los que dependen de User
        ProductSeeder::class,
        // ... otros seeders
    ]);

    $this->command->info('¡Sembrado completado!');
}
```

Módulo 3: Relaciones Avanzadas en la Base de Datos

Eloquent permite definir y trabajar con relaciones entre modelos de una manera muy intuitiva.

3.1. Relaciones Uno a Muchos (1:N)

Un `User` tiene muchos `Post`. Un `Post` pertenece a un `User`.

```
// app/Models/User.php
public function posts()
{
    return $this->hasMany(Post::class);
}

// app/Models/Post.php
public function user()
{
    return $this->belongsTo(User::class);
}
```

3.2. Relaciones Muchos a Muchos (N:M)

Un `Post` puede publicarse en muchos `Channel`, y un `Channel` puede tener muchos `Post`.

```
// app/Models/Post.php
public function channels()
{
    // El segundo argumento es el nombre de la tabla pivote
    return $this->belongsToMany(Channel::class, 'post_channels');
}

// app/Models/Channel.php
public function posts()
{
    return $this->belongsToMany(Post::class, 'post_channels');
}
```

3.3. Tablas Pivote: El Corazón de las Relaciones N:M

Para que una relación N:M funcione, se necesita una tabla intermedia, llamada **tabla pivote**. Su migración típicamente contiene las claves foráneas de las dos tablas que conecta.

Migración de la tabla `post_channels`:

```
Schema::create('post_channels', function (Blueprint $table) {
    $table->id();
    $table->foreignId('post_id')->constrained()->onDelete('cascade');
    $table->foreignId('channel_id')->constrained()->onDelete('cascade');
    $table->timestamps();

    // Evita duplicados: un post no puede estar dos veces en el mismo canal
    $table->unique(['post_id', 'channel_id']);
});
```

> **Corrección común:** Al usar `constrained()`, Laravel infiere el nombre de la tabla a partir del nombre de la columna (ej. `media_id` -> tabla `media`). Si tu tabla se llama en plural (`medias`), debes especificarlo: `$table->foreignId('media_id')->constrained('medias')`.

3.4. Trabajando con Datos de Tablas Pivote

Eloquent proporciona métodos para gestionar estas relaciones fácilmente:

- `attach($id)`: Añade una nueva relación.
- `detach($id)`: Elimina una relación.
- `sync([1, 2, 3])`: Sincroniza las relaciones. Elimina las que no estén en el array y añade las nuevas.
- `withPivot(['columna1', 'columna2'])`: Permite acceder a columnas adicionales en la tabla pivote.

Módulo 4: El Corazón de la Aplicación: Rutas y Controladores

4.1. El Sistema de Enrutamiento de Laravel

Las **Rutas** son el punto de entrada de la aplicación. Mapean una URL y un método HTTP (GET, POST, etc.) a una acción específica.

- `routes/web.php` : Para rutas de la interfaz web tradicional.
- `routes/api.php` : Para rutas de la API, que son stateless y se prefijan automáticamente con `/api` .

4.2. Tipos de Rutas: Básicas, con Parámetros y Nombradas

```
// routes/web.php
use App\Http\Controllers\ProductController;

// Ruta básica a una vista
Route::get('/', function () {
    return view('welcome');
});

// Ruta a un método de un controlador
Route::get('/products', [ProductController::class, 'index']);

// Ruta con un parámetro obligatorio
Route::get('/products/{id}', [ProductController::class, 'show']);

// Ruta nombrada para fácil referencia
Route::get('/admin/dashboard', [AdminController::class, 'dashboard'])->name('admin.dashboard');
```

4.3. Controladores: Organizando la Lógica

Los **Controladores** agrupan la lógica de manejo de peticiones HTTP en una sola clase, manteniendo el código limpio y organizado.

Para crear un controlador:

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

4.4. Controladores de Tipo Recurso (RESTful)

Para operaciones CRUD estándar, Laravel ofrece los controladores de tipo recurso, que siguen las convenciones REST.

```
php artisan make:controller ProductController --resource --model=Product
```Este comando crea un controlador con los siguientes métodos:
*   `index()`: Mostrar una lista de recursos.
*   `create()`: Mostrar el formulario para crear un nuevo recurso.
*   `store()`: Almacenar un nuevo recurso.
*   `show(Product $product)`: Mostrar un recurso específico.
*   `edit(Product $product)`: Mostrar el formulario para editar un recurso.
*   `update(Request $request, Product $product)`: Actualizar un recurso.
*   `destroy(Product $product)`: Eliminar un recurso.

Se registra en el archivo de rutas con una sola línea:
```php
Route::resource('products', ProductController::class);
```

Módulo 5: Creando una API REST

5.1. Diferencias entre Controladores Web y API

Aspecto	Controladores Web	Controladores API
Respuesta	Vistas HTML (`return view(...)`)	JSON (`return response()->json(...)`)
Estado	Con estado (usa sesiones/cookies)	Sin estado (Stateless)
Autenticación	Sesiones	Tokens (ej. Sanctum)
Errores	Redirección con mensajes de error	Códigos de estado HTTP (404, 422, 500)

5.2. Creando Rutas y Controladores para la API

Las rutas de la API se definen en `routes/api.php` .

Para crear un controlador optimizado para API:```bash
php artisan make:controller Api/ProductController --api

Esto omite los métodos `create()` y `edit()`, que son para formularios HTML.

Ejemplo de método en un controlador de API:

```php
// app/Http/Controllers/Api/ProductController.php
public function index()
{
    // Retorna una colección de productos en formato JSON
    return Product::all();
}
```

5.3. Integración con un Frontend Básico (Vanilla JS)

Una API puede ser consumida por cualquier cliente (una app móvil, otra aplicación, o un frontend en JavaScript). Usando la **Fetch API** de JavaScript, podemos comunicarnos con nuestra API de Laravel.

Ejemplo `app.js`:

```javascript
const API_URL = 'http://localhost:8000/api';

async function fetchProducts() {
    try {
        const response = await fetch(`${API_URL}/products`);
        if (!response.ok) {
            throw new Error('La respuesta de la red no fue correcta');
        }
        const products = await response.json();
        console.log(products); // Aquí renderizarías los productos en el HTML
    } catch (error) {
        console.error('Hubo un problema con la petición Fetch:', error);
    }
}

fetchProducts();
```

Módulo 6: Autenticación y Autorización de APIs

6.1. Autenticación con Laravel Sanctum

Laravel Sanctum es un sistema ligero de autenticación para SPAs (Single Page Applications) y APIs basadas en tokens.

Instalación y Configuración:

1. Instalar el paquete: `composer require laravel/sanctum`
2. Publicar la configuración y migración:
   `php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"`
3. Ejecutar la migración: `php artisan migrate`
4. Añadir el trait `HasApiTokens` a tu modelo `User`.

```
// app/Models/User.php
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable, HasRoles;
    // ...
}
```

6.2. Emisión y Gestión de Tokens de API

Cuando un usuario inicia sesión, le generamos un token. Este token deberá ser enviado en cada petición subsecuente para autenticar al usuario.

Lógica de Login y creación de token:

```php
// En tu AuthController
use Illuminate\Support\Facades\Hash;

public function login(Request $request)
{
    $user = User::where('email', $request->email)->first();

    if (!$user || !Hash::check($request->password, $user->password)) {
        return response()->json(['message' => 'Credenciales incorrectas'], 401);
    }

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

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

6.3. Protección de Rutas con Middleware

Para proteger rutas y requerir un token válido, usamos el middleware `auth:sanctum` .

```
// routes/api.php
Route::middleware('auth:sanctum')->group(function () {
    // Estas rutas solo serán accesibles con un token válido
    Route::get('/user', function (Request $request) {
        return $request->user();
    });
    Route::post('/logout', [AuthController::class, 'logout']);
});```

El cliente debe enviar el token en el header `Authorization` de cada petición: `Authorization: E

### 6.4. Autorización con Spatie Laravel-Permission

Una vez que un usuario está autenticado, necesitamos saber *qué puede hacer*. El paquete **Spati

**Instalación y Configuración:**
1. Instalar: `composer require spatie/laravel-permission`
2. Publicar y migrar: `php artisan vendor:publish --provider="Spatie\Permission\PermissionServi
3. Añadir el trait `HasRoles` al modelo `User`.

**Uso básico:**
```php
// Crear roles y permisos (generalmente en un seeder)
$adminRole = Role::create(['name' => 'admin']);
$editPermission = Permission::create(['name' => 'edit articles']);
$adminRole->givePermissionTo($editPermission);

// Asignar un rol a un usuario
$user->assignRole('admin');

// Proteger una ruta con un permiso o rol
Route::post('/articles', ...)->middleware('permission:edit articles');
Route::get('/admin/panel', ...)->middleware('role:admin');
```
```

Apéndice: Comandos Artisan Útiles

```
# Migraciones {#migraciones }
php artisan make:migration <nombre_migracion>
php artisan migrate
php artisan migrate:rollback
php artisan migrate:fresh --seed  # Borra todo y ejecuta migraciones + seeders

# Modelos {#modelos }
php artisan make:model <NombreModelo>
php artisan make:model <NombreModelo> -m # Crea migración
php artisan make:model <NombreModelo> -f # Crea factory
php artisan make:model <NombreModelo> -c # Crea controlador
php artisan make:model <NombreModelo> -s # Crea seeder
php artisan make:model <NombreModelo> -mfcs # Crea todo

# Controladores {#controladores }
php artisan make:controller <NombreControlador>
php artisan make:controller <NombreControlador> --resource
php artisan make:controller Api/<NombreControlador> --api

# Rutas {#rutas }
php artisan route:list # Muestra todas las rutas de la aplicación

# Seeders y Factories {#seeders-y-factories }
php artisan make:seeder <NombreSeeder>
php artisan make:factory <NombreFactory>
php artisan db:seed
php artisan db:seed --class=<NombreSeeder>
```