Video final

Video

Manga Store - Diario de Desarrollo

Resumen por Día

Día 1: Creación del proyecto Laravel, modelos (Cliente y Pedido), migraciones y controladores CRUD. Factories y seeders para datos de prueba.

Día 2: Definición de relaciones entre modelos (`hasMany` y `belongsTo`). Implementación de factories con Faker. Seeder para llenar la base de datos automáticamente. Rutas con `Route::resource`.

Día 3: Uso de `HasFactory` en modelos. Comando `migrate:fresh --seed`. Implementación de métodos `index()` y `show()` en controladores. Introducción a Eager Loading con `with()`.

Día 4: Completación del CRUD en ambos controladores. Sistema de validaciones. Route Model Binding. Métodos `create()`, `store()`, `edit()`, `update()`, `destroy()`.

Cosas aprendidas:

Dia 1

El atajo "-mcr": Creando todo de un golpe

¿Para qué sirve? : Con este añadido al final del comando, Laravel te construye la estructura básica de una sección de tu web de un solo golpe. Te crea tres archivos clave:

- **La M (Migración):** El plano para crear la tabla en la base de datos.
- **La C (Controlador):** El "cerebro" que recibe las peticiones de los usuarios.
- **La R (Recurso):** Hace que ese controlador ya venga con los métodos estándar para ver, crear, editar y borrar (el famoso CRUD) ya escritos, para que no tengas que crearlos tú uno a uno.

Uso real: Se utiliza para **ahorrar tiempo y evitar errores de nombres** .

Dia 2

`hasMany` (En el modelo Cliente)

¿Para qué sirve? : Para que cuando estemos con un cliente, podamos sacar de golpe todos sus pedidos.

Uso real : `$cliente->pedidos` (nos devuelve la lista de sus compras).

`belongsTo` (En el modelo Pedido)

¿Para qué sirve? : Para ponerle nombre y apellidos a cada pedido.

Uso real : `$pedido->cliente` (nos devuelve los datos del cliente que hizo esa compra).

Faker: Datos de prueba

¿Para qué sirve? : Genera datos falsos (nombres, emails) automáticamente para no tener que inventarlos y escribirlos tú a mano. **Uso real** : Rellena la base de datos en un segundo para probar que el diseño de la web funciona bien con mucha información.

Route::resource tiene 7 rutas

¿Para qué sirve? : Crea de golpe las 7 rutas estándar para gestionar algo (ver lista, crear, guardar, ver detalle, editar, actualizar y borrar) con una sola línea. **Uso real** : Permite tener todas las funciones de "crear, leer, editar y borrar" (CRUD) listas y ordenadas sin tener que escribir cada ruta por separado.

Concretamente son estas 7 usando la entidad cliente:

Las 7 rutas automáticas

Dirección (URL)	Método	Qué hace (en sencillo)
`/clientes`	GET	Muestra la lista de todos los clientes.
`/clientes/create`	GET	Abre el formulario para escribir los datos del nuevo cliente.
`/clientes`	POST	Es la dirección interna donde se mandan los datos para guardarlos.
`/clientes/{id}`	GET	Muestra la ficha de un solo cliente (ej:`/clientes/5`).
`/clientes/{id}/edit`	GET	Abre el formulario con los datos ya rellenos para cambiarlos.
`/clientes/{id}`	PUT	Es la dirección que procesa los cambios que has editado.
`/clientes/{id}`	DELETE	La dirección que se encarga de borrar a ese cliente.

Dia 3

migrate:fresh

```
php artisan migrate:fresh --seed
```

¿Para qué sirve? : El comando ejecuta una secuencia destructiva-constructiva: primero ejecuta un `DROP ALL TABLES` (eliminando tablas, índices y vistas sin importar las restricciones de integridad), luego recompila el esquema SQL ejecutando cronológicamente todos los métodos `up()` de tus migraciones, y finalmente dispara el `DatabaseSeeder` para realizar operaciones de `INSERT` masivas.

Uso real : Se utiliza principalmente en entornos de desarrollo cuando el esquema de la base de datos se ha vuelto "pesado" o inconsistente debido a cambios constantes en los requerimientos.

> ❗ **PELIGRO: IMPACTO CRÍTICO** Impacto Crítico: Este comando ejecuta un DROP ALL TABLES antes de migrar. Úsalo solo en desarrollo local. Ejecutarlo en producción borrará permanentemente los datos de los clientes y la estructura de la base de datos.

witch (Eager Loading)

¿Para qué sirve? : Para cargar relaciones de forma Eager. Su función técnica es indicarle a Laravel que debe traer los datos del modelo relacionado en una única consulta SQL (usando un `JOIN` o un `IN`), evitando así el problema de las consultas N+1 .

Uso real : `$pedidos = Pedido::with('cliente')->get();` (En la lista de ventas de MangaStore, permite mostrar el nombre de cada lector junto a su pedido de manga realizando solo 2 consultas a la base de datos, en lugar de hacer una consulta nueva por cada pedido que aparezca en la lista).

Diario de Trabajo: Dia 1

Cración del proyecto

Primero creamos el proyecto y entramos a el con los comandos:

```
composer create-project laravel/laravel manga-store
cd manga-store
```

Creación del modelo, Migracion y Controladores

Luego creamos el modelo, migraciones y controladores.

```
php artisan make:model Cliente -mcr
php artisan make:model Pedido -mcr
```

> Nota:
>
> `-mcr` crea el Modelo, La Migración y el Controlador con los métodos CRUD ya definidos

Creación Factories y Seeders

```
php artisan make:factory ClienteFactory
php artisan make:factory PedidoFactory
php artisan make:seeder ClienteSeeder
```

Migraciones (Database)

create_clientes_table

Esta tabla almacena la información básica de los usuarios que realizarán compras en la tienda.

- `id()` : Crea un campo autoincremental como clave primaria.
- `nombre` y `telefono` : Campos de texto estándar para identificación y contacto.
- `email` : Se marca como `unique()` para evitar que dos clientes se registren con el mismo correo.
- `direccion` : Se usa `text` en lugar de `string` para permitir descripciones largas de domicilio.
- `activo` : Un booleano que permite "desactivar" clientes sin borrar sus datos (borrado lógico).
- `timestamps()` : Crea automáticamente las columnas `created_at` y `updated_at`.

```
public function up(): void {
    Schema::create('clientes', function (Blueprint $table) {
        $table->id();
        $table->string('nombre');
        $table->string('email')->unique();
        $table->string('telefono');
        $table->text('direccion');
        $table->boolean('activo')->default(true);
        $table->timestamps();
    });
}
```

create_pedidos_table

Esta tabla gestiona las transacciones de la tienda y vincula cada compra con un cliente específico.

- `numero_pedido` : Un identificador único para seguimiento comercial (diferente al ID interno).
- `fecha` : Registra el momento exacto de la venta.
- `estado` : Utiliza un `enum`, lo que restringe los valores posibles a solo cuatro opciones específicas, garantizando la integridad de los datos.
- `total` : Definido como `decimal(8, 2)` para manejar dinero con precisión (evitando los errores de redondeo de los tipos *float*).
- `foreignId('cliente_id')` : Es la pieza clave de la relación. Conecta el pedido con un ID de la tabla `clientes`.
- `constrained()` : Asegura que el cliente realmente exista.

- `onDelete('cascade')` : Si un cliente es eliminado de la base de datos, todos sus pedidos se borrarán automáticamente para no dejar datos huérfanos.

```
public function up(): void {
    Schema::create('pedidos', function (Blueprint $table) {
        $table->id();
        $table->string('numero_pedido')->unique(); // Ej: MANGA-1001
        $table->date('fecha');
        $table->enum('estado', ['pendiente', 'enviado', 'entregado', 'cancelado'])-
>default('pendiente');
        $table->decimal('total', 8, 2);
        $table->text('notas')->nullable();
        // Relación: Un pedido pertenece a un cliente
        $table->foreignId('cliente_id')->constrained()->onDelete('cascade');
        $table->timestamps();
    });
}
```

Diario de Trabajo: Dia 2

Modelos y Relaciones

En esta parte definimos cómo se comportan nuestros datos y cómo se conectan entre ellos.

Modelo Cliente

```
class Cliente extends Model {
    protected $fillable = ['nombre', 'email', 'telefono', 'direccion', 'activo'];

    // Un cliente tiene muchos pedidos
    public function pedidos() {
        return $this->hasMany(Pedido::class);
    }
}
```

- `$fillable` : Es una medida de seguridad. Aquí ponemos los campos que permitimos que se rellenen.
- **Relación** `pedidos()` : Aquí le decimos a Laravel que un cliente puede tener muchos pedidos. Gracias a esto, luego podremos sacar los pedidos de alguien

con un simple `$cliente->pedidos`.

Modelo Pedido

```
class Pedido extends Model {
    protected $fillable = ['numero_pedido', 'fecha', 'estado', 'total', 'notas',
'cliente_id'];

    // Un pedido pertenece a un cliente
    public function cliente() {
        return $this->belongsTo(Cliente::class);
    }
}
```

Relación `cliente()` : Es la inversa de la anterior. Cada pedido pertenece a un único cliente. Esto nos permite saber quién hizo la compra fácilmente.

Factories y Seeders

Para no tener que estar metiendo datos a mano cada vez que probamos la web, usamos estas herramientas para generar "relleno" automático.

ClienteFactory

```
public function definition(): array {
    return [
        'nombre' => $this->faker->name(),
        'email' => $this->faker->unique()->safeEmail(),
        'telefono' => $this->faker->phoneNumber(),
        'direccion' => $this->faker->address(),
        'activo' => true,
    ];
}
```

Usamos la librería **Faker** , que es como un generador de identidades falsas. Nos crea nombres, emails, teléfonos y direcciones que parecen reales pero son aleatorios. El campo `activo` lo dejamos en `true` por defecto.

PedidoFactory

```
 public function definition(): array
    {
        return [
            'numero_pedido' => 'MNGA-' . $this->faker->unique()-
>numberBetween(1000, 9999),
            'fecha' => now(),
            'estado' => $this->faker->randomElement(['pendiente', 'enviado',
'entregado']),
            'total' => $this->faker->randomFloat(2, 10, 200),
            'notas' => $this->faker->sentence(),
        ];
    }
```

DatabaseSeeder

```
public function run(): void {
    Cliente::factory(10)->create()->each(function ($cliente) {
        Pedido::factory(3)->create([
            'cliente_id' => $cliente->id,
        ]);
    });
}
```

Este es el "botón de encendido" de nuestros datos de prueba. Lo que hace el código es:

1. Crear 10 clientes ficticios.
2. Para cada uno de esos clientes, crear 3 pedidos automáticamente. Así, nada más empezar, tenemos una base de datos con 30 pedidos listos para mostrar.

Rutas

Web.php

```
use App\Http\Controllers\ClienteController;
use App\Http\Controllers\PedidoController;
```

```
Route::resource('clientes', ClienteController::class);
Route::resource('pedidos', PedidoController::class);
```

`Route::resource`: Esta línea es un "atajo" que crea automáticamente las 7 rutas típicas de un CRUD (Index, Create, Store, Show, Edit, Update, Destroy). Es mucho más limpio y sigue el estándar de Laravel.

Diario de Trabajo: Dia 3

Modificaciones del model

lo que vamos a añadir a los dos modelos es encima del `$fillable` una pequeña linea para usar el factory: `use HasFactory;`

Comandos por terminal

```
# Refrescar la base de datos y ejecutar los seeders
php artisan migrate:fresh --seed

# Para crear las carpetas de las vistas
mkdir resources\views\clientes
mkdir resources\views\pedidos
mkdir resources\views\layouts
```

El primer comando su funcion principal es borrar la base de datos y crear la base de datos con los datos que esten en migrate. esto es muy util pero tambien muy peligroso si se usa por error porque borra todo de golpe.

Los otros comandos son para crear las capetas de las vistas

> ⚠️ Aclaración
>
> Los blade.php los e creado a mano por eso no saldra el `php artisan make:view`

ClienteController.php

```php
public function index() {
    $clientes = Cliente::all();
    return view('clientes.index', compact('clientes'));
}

public function show(Cliente $cliente) {
    return view('clientes.show', compact('cliente'));
}
```

He optado por programar únicamente los métodos `index` y `show` por una razón de seguridad técnica: validar la capa de persistencia . Antes de construir formularios complejos para crear o editar datos.

PedidoController.php

```php
public function index() {
    // Usamos 'with' para que la consulta sea más eficiente
    $pedidos = Pedido::with('cliente')->get();
    return view('pedidos.index', compact('pedidos'));
}
```

He implementado únicamente el método `index` para validar la eficiencia de las relaciones mediante Eager Loading .

Diario de Trabajo: Dia 4

En este día completamos los controladores para tener un CRUD funcional completo. Implementamos tanto en `ClienteController` como en `PedidoController` todos los métodos necesarios para gestionar el ciclo de vida de la información.

ClienteController Completado

Método `create()`

```php
public function create() {
    return view('clientes.create');
```

```
}
```

Simplemente abre el formulario vacío para que el usuario pueda escribir los datos de un nuevo cliente.

Método `store()`

```
public function store(Request $request) {
    $request->validate([
        'nombre' => 'required|string|max:255',
        'email' => 'required|email|unique:clientes,email',
        'telefono' => 'required',
        'direccion' => 'required',
    ]);

    Cliente::create($request->all());
    return redirect()->route('clientes.index')->with('success', 'Lector registrado
con éxito.');
}
```

- `validate()` : Verifica que los datos cumplen las reglas. Si no, rechaza automáticamente la petición.
- `unique:clientes,email` : Asegura que no hay otro cliente con ese email.
- `Cliente::create()` : Guarda directamente en la base de datos.
- `redirect()->route()` : Redirige a la lista de clientes después de guardar.
- `with('success',...)` : Envía un mensaje de éxito a la vista.

Método `edit()`

```
public function edit(Cliente $cliente) {
    return view('clientes.edit', compact('cliente'));
}
```

Abre el formulario pero con los datos del cliente ya rellenos (gracias al Route Model Binding).

Método `update()`

```
public function update(Request $request, Cliente $cliente) {
    $request->validate([
        'nombre' => 'required',
        'email' => 'required|email|unique:clientes,email,' . $cliente->id,
    ]);

    $cliente->update($request->all());
    return redirect()->route('clientes.index')->with('success', 'Datos del lector
actualizados.');
}
```

- `unique:clientes,email,' . $cliente->id` : Permite que el email sea el mismo si pertenece al mismo cliente. Si el usuario cambia el email, verifica que el nuevo no esté en uso.
- `$cliente->update()` : Actualiza solo los campos que vienen en el request.

Método `destroy()`

```
public function destroy(Cliente $cliente) {
    $cliente->delete();
    return redirect()->route('clientes.index')->with('success', 'Lector
eliminado.');
}
```

Borra el cliente y redirige a la lista. Si tiene pedidos asociados, se borran automáticamente por la regla `onDelete('cascade')` que pusimos en la migración.

PedidoController Completado

Método `create()`

```
public function create() {
    $clientes = Cliente::all(); // Traemos todos los lectores para el select
    return view('pedidos.create', compact('clientes'));
}
```

Abre el formulario para crear un nuevo pedido. Trae la lista de clientes para que el usuario seleccione a quién pertenece el pedido.

Método `store()`

```
public function store(Request $request) {
    $validated = $request->validate([
        'numero_pedido' => 'required|unique:pedidos,numero_pedido',
        'cliente_id' => 'required|exists:clientes,id',
        'total' => 'required|numeric|min:0',
        'fecha' => 'required|date',
        'estado' => 'required|in:pendiente,enviado,entregado,cancelado',
    ]);

    Pedido::create($validated);

    return redirect()->route('pedidos.index')
        ->with('success', '¡El nuevo pedido ha sido registrado correctamente!');
}
```

- `numero_pedido` : Debe ser único (no pueden haber dos pedidos con el mismo número).
- `exists:clientes,id` : Valida que el cliente seleccionado realmente existe en la base de datos.
- `in:pendiente,enviado,entregado,cancelado` : Solo permite estos 4 estados posibles.
- `$validated` : Guardamos los datos validados en una variable para mayor seguridad.

Método `show()`

```
public function show(Pedido $pedido) {
    return view('pedidos.show', compact('pedido'));
}
```

Muestra los detalles de un pedido específico. Gracias a la relación `belongsTo`, podemos acceder al cliente del pedido con `$pedido->cliente` en la vista.

Método `edit()`

```
public function edit(Pedido $pedido) {
    return view('pedidos.edit', compact('pedido'));
}
```

Abre el formulario de edición con los datos del pedido ya rellenos.

Método `update()`

```
public function update(Request $request, Pedido $pedido) {
    $validated = $request->validate([
        'total' => 'required|numeric|min:0',
        'fecha' => 'required|date',
        'estado' => 'required|in:pendiente,enviado,entregado,cancelado',
    ]);

    $pedido->update($validated);

    return redirect()->route('pedidos.index')->with('success', 'Pedido actualizado
correctamente.');
}
```

Actualiza el pedido. Notamos que NO validamos `numero_pedido` ni `cliente_id` porque esos datos no deben cambiar una vez creados (son identificadores). Solo se pueden editar el total, la fecha y el estado.

Método `destroy()`

```
public function destroy(Pedido $pedido) {
    $pedido->delete();
    return redirect()->route('pedidos.index')->with('success', 'Pedido eliminado
del sistema.');
}
```

Borra el pedido de la base de datos y redirige a la lista.

Pantalla de Inicio

En `routes/web.php` añadimos una ruta raíz para la pantalla de bienvenida:

```php
Route::get('/', function () {
    return view('welcome');
});
```

Cuando el usuario accede a `http://localhost/`, se muestra la vista `welcome.blade.php`. Esta es la puerta de entrada a nuestra aplicación.