UD.2 Práctica Guiada: Rutas, Controladores y Vistas

(Prácticas de Guillermo Garrido modificadas por Eva María Gómez Abad - Curso 2025-2026)

Objetivo

El objetivo de esta práctica es construir una tienda online completa utilizando Laravel, implementando el patrón MVC (Modelo-Vista-Controlador). Se crearán rutas públicas, controladores para manejar la lógica de negocio y vistas Blade para mostrar los datos de forma dinámica.

Filosofía de Trabajo

Esta práctica está diseñada para ser genérica y personalizable. A partir de la temática elegida en la práctica anterior se adaptaran todos los nombres, categorías y productos según el tipo de tienda.

FASE 1: Iniciar el Entorno de Desarrollo

1. Arrancar los Contenedores de Laravel Sail

Antes de comenzar a trabajar en el código, es fundamental que el entorno de desarrollo esté funcionando correctamente.

Asegúrate de que Laravel Sail esté ejecutándose:

```
sail up -d
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ sail up -d
[+] Running 7/7
 ✔ sail-8.4/app                    Built                          0.0s
 ✔ Network myshop_sail             Created                        0.0s
 ✔ Volume myshop_sail-redis        Created                        0.0s
 ✔ Volume myshop_sail-mysql        Created                        0.0s
 ✔ Container myshop-mysql-1        Started                        0.8s
 ✔ Container myshop-redis-1        Started                        0.8s
 ✔ Container myshop-laravel.test-1 Started                        1.0s
```

2. Verificar que la Aplicación Funciona

Abre tu navegador y accede a `http://localhost`. Deberías ver la página de bienvenida de Laravel que creaste en la práctica anterior.

> [!NOTE] Importante Si los contenedores no arrancan correctamente, revisa que Docker esté ejecutándose y que no haya otros servicios usando los puertos 80, 3306 o 6379.

FASE 2: Crear los Archivos de Datos Mock

1. Crear el Directorio de Datos Mock

Hemos decidido estructurar los datos mock como un **array de arrays** (una **Matriz**) en PHP porque esta estructura es sencilla y fácil de manipular. Cada elemento del array principal representa una entidad (por ejemplo, una categoría o un producto) y se identifica por su clave numérica o por su ID. Dentro de cada elemento, se utiliza un array asociativo para almacenar los diferentes atributos de la entidad (como `id`, `name`, `description`, etc.). Esta forma de organización facilita la lectura, la búsqueda y la modificación de los datos mock durante el desarrollo.

En este proyecto, vamos a crear varios archivos de datos mock para centralizar la información de prueba. Los principales mocks que vamos a crear son:

- **mock-categories.php**: Contendrá las categorías de productos.
- **mock-products.php**: Incluirá los productos de la tienda.
- **mock-offers.php**: Definirá las ofertas disponibles para los productos.
- **mock-cart.php**: Simulará el estado del carrito de compras.

Todos estos archivos estarán ubicados en el directorio `database/data/` para mantenerlos organizados y accesibles.

Crea el directorio `database/data/`:

```
mkdir -p database/data
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p database/data
ggarrido@mi-equipo:~/development/laravel/MyShop$ ls -la database/
total 20
drwxrwxr-x 4 ggarrido ggarrido 4096 ene 15 10:30 .
drwxrwxr-x 8 ggarrido ggarrido 4096 ene 15 10:30 ..
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:30 data
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:30 factories
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:30 migrations
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:30 seeders
```

2. Crear el Archivo de Categorías

Los diferentes atributos de una categoría suelen ser:

- **id**: Identificador único de la categoría.
- **name**: Nombre de la categoría.
- **slug**: Identificador amigable para URLs (por ejemplo, "electronica", "ropa-hombre").
- **description**: Descripción breve de la categoría.

Opcionalmente, se pueden añadir otros atributos como:

- **image**: Ruta o URL de una imagen representativa de la categoría.
- **parent_id**: Si la categoría es hija de otra, este campo indica la categoría padre.

- **created_at / updated_at**: Fechas de creación y actualización (si se requiere control de cambios).

Crea el archivo `database/data/mock-categories.php`:

```
touch database/data/mock-categories.php
```

Añade el código:

```php
<?php

return [
    1 => [
        'id' => 1,
        'name' => 'Categoría 1',
        'slug' => 'categoria-1',
        'description' => 'Descripción de la categoría 1'
    ],
    2 => [
        'id' => 2,
        'name' => 'Categoría 2',
        'slug' => 'categoria-2',
        'description' => 'Descripción de la categoría 2'
    ],
    3 => [
        'id' => 3,
        'name' => 'Categoría 3',
        'slug' => 'categoria-3',
        'description' => 'Descripción de la categoría 3'
    ],
    4 => [
        'id' => 4,
        'name' => 'Categoría 4',
        'slug' => 'categoria-4',
        'description' => 'Descripción de la categoría 4'
    ],
    5 => [
        'id' => 5,
        'name' => 'Categoría 5',
        'slug' => 'categoria-5',
        'description' => 'Descripción de la categoría 5'
    ]
];
```

3. Crear el Archivo de Ofertas

Los atributos principales de una oferta suelen ser:

- **id**: Identificador único de la oferta.
- **name**: Nombre de la oferta (por ejemplo, "Descuento Primavera").

- **slug**: Identificador amigable para URLs (por ejemplo, "descuento-primavera").
- **discount_percentage**: Porcentaje de descuento que aplica la oferta (por ejemplo, 20 para un 20% de descuento).
- **description**: Descripción breve de la oferta, explicando sus condiciones o beneficios.

Opcionalmente, puedes añadir otros atributos como:

- **start_date**: Fecha de inicio de la oferta.
- **end_date**: Fecha de finalización de la oferta.
- **active**: Indica si la oferta está activa o no.
- **product_ids**: Lista de IDs de productos a los que aplica la oferta.

Crea el archivo `database/data/mock-offers.php`:

```
touch database/data/mock-offers.php
```

Añade el código:

```
<?php
//ToDo
;
```

4. Crear el Archivo de Carrito (Cart Items)

En esta práctica, vamos a simular un carrito de compras sencillo donde todos los items pertenecen a un único usuario/sesión global. Esto simplifica la lógica y permite centrarnos en el patrón MVC sin añadir complejidad de gestión de usuarios.

Estructura del carrito:

- **id**: Identificador único del item en el carrito
- **product_id**: ID del producto añadido al carrito
- **quantity**: Cantidad de unidades de ese producto
- **added_at**: Fecha y hora en la que se añadió el producto

> [!NOTE] Importante En esta práctica NO gestionamos usuarios ni sesiones. En la práctica de autenticación se añadirá `user_id` para gestionar carritos de múltiples usuarios.

Crea el archivo `database/data/mock-cart.php`:

```
touch database/data/mock-cart.php
```

Añade el código:

```

```

```php
<?php

// Simulación de items en el carrito (carrito único/global)
// Todos los items pertenecen al mismo carrito
return [
    1 => [
        'id' => 1,
        'product_id' => 1,
        'quantity' => 2,
        'added_at' => '2025-01-15 10:30:00'
    ],
    2 => [
        'id' => 2,
        'product_id' => 3,
        'quantity' => 1,
        'added_at' => '2025-01-15 11:15:00'
    ],
    3 => [
        'id' => 3,
        'product_id' => 5,
        'quantity' => 4,
        'added_at' => '2025-01-15 12:00:00'
    ]
];
```

5. Crear el Archivo de Productos

A continuación se explican los atributos que debe tener cada producto en el archivo `mock-products.php`:

- **id**: Identificador único del producto
- **name**: Nombre del producto que se mostrará en la tienda
- **description**: Breve descripción del producto
- **price**: Precio del producto (puede ser decimal)
- **category_id**: Identificador de la categoría a la que pertenece el producto
- **offer_id**: ID de la oferta aplicable a este producto (puede ser `null` si no tiene oferta)

Puedes añadir más atributos personalizados según la temática de tu tienda, por ejemplo: autor, ISBN, talla, color, marca, stock, imagen, etc.

> [!NOTE] Reglas de negocio importantes
>
> - **Una oferta** puede aplicarse a **múltiples productos** (relación 1:N)
> - **Un producto** solo puede tener **una oferta activa** a la vez (o ninguna)
> - No todos los productos tienen ofertas (en ese caso `offer_id` es `null`)

Crea el archivo `database/data/mock-products.php`:

```
touch database/data/mock-products.php
```

Añade el código:

```
<?php

return [
    1 => [
        'id' => 1,
        'name' => 'Producto 1',
        'description' => 'Descripción del producto 1',
        'price' => 29.99,
        'category_id' => 1,
        'offer_id' => 1  // Tiene la oferta 1 (20% descuento)
    ],
    //ToDo.Completa tus ofertas
];
```

> [!CAUTION] Captura de pantalla requerida Realiza una captura de pantalla de tus archivos de datos mock (`database/data/products.php`, `categories.php`, `offers.php`, etc.) **ajustados a la temática que elegiste** (por ejemplo, libros, ropa, electrónica, etc.). La captura debe mostrar claramente cómo personalizaste los nombres, descripciones y campos de los productos, categorías y ofertas según tu temática. Incluye tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible.

FASE 3: Crear el Trait para Cargar Datos Mock

1. Crear el Trait LoadsMockData

En esta sección vamos a crear un Trait llamado `LoadsMockData` que nos permitirá cargar fácilmente los datos simulados (mock) de categorías, ofertas, productos y carrito desde los archivos PHP ubicados en `database/data/`.

Este Trait será utilizado en los controladores para acceder a los datos de ejemplo sin necesidad de una base de datos real, facilitando así el desarrollo y las pruebas de la tienda online.

El objetivo es centralizar la carga de datos mock y reutilizar este código en cualquier controlador que lo necesite.

Además, este Trait incluirá un método helper llamado `enrichProductsWithOffers()` que será fundamental para preparar los datos antes de enviarlos a las vistas. Este método:

1. **Enriquece cada producto** con los datos completos de su oferta (nombre, porcentaje de descuento)
2. **Calcula el precio final** aplicando el descuento correspondiente
3. **Evita hardcodear valores** en las vistas y componentes

De esta forma, el componente `ProductCard` recibirá toda la información necesaria sin necesidad de hacer cálculos ni búsquedas en la vista.

Crea el directorio `app/Traits/`:

```
mkdir -p app/Traits
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p app/Traits
ggarrido@mi-equipo:~/development/laravel/MyShop$ ls -la app/
total 20
drwxrwxr-x 8 ggarrido ggarrido 4096 ene 15 10:35 .
drwxrwxr-x 8 ggarrido ggarrido 4096 ene 15 10:35 ..
drwxrwxr-x 3 ggarrido ggarrido 4096 ene 15 10:35 Console
drwxrwxr-x 3 ggarrido ggarrido 4096 ene 15 10:35 Exceptions
drwxrwxr-x 3 ggarrido ggarrido 4096 ene 15 10:35 Http
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:35 Models
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:35 Providers
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:35 Traits
```

Crea el archivo `app/Traits/LoadsMockData.php`:

```
touch app/Traits/LoadsMockData.php
```

Añade el código:

```php
<?php

namespace App\Traits;

trait LoadsMockData
{
    /**
     * Load categories from mock file
     */
    protected function getCategories(): array
    {
        return require database_path('data/mock-categories.php');
    }

    /**
     * Load offers from mock file
     */
    protected function getOffers(): array
    {
        return require database_path('data/mock-offers.php');
    }

    /**
     * Load cart from mock file
     */
    protected function getCart(): array
    {
        return require database_path('data/mock-cart.php');
```

```
    }

    /**
     * Load products from mock file
     */
    protected function getProducts(): array
    {
        return require database_path('data/mock-products.php');
    }

    /**
     * Load all mock data at once
     */
    protected function getAllMockData(): array
    {
        return [
            'categories' => $this->getCategories(),
            'offers' => $this->getOffers(),
            'cart' => $this->getCart(),
            'products' => $this->getProducts(),
        ];
    }

    /**
     * Enrich products with their offer data and calculate final price
     * This method adds 'offer' and 'final_price' to each product that has an
offer
     */
    protected function enrichProductsWithOffers(array $products): array
    {
        $offers = $this->getOffers();

        return array_map(function($product) use ($offers) {
            // Add offer data if product has an offer
            if ($product['offer_id'] !== null &&
isset($offers[$product['offer_id']])) {
                $offer = $offers[$product['offer_id']];
                $product['offer'] = $offer;

                // Calculate final price with discount
                $discount = $product['price'] * ($offer['discount_percentage'] /
100);
                $product['final_price'] = $product['price'] - $discount;
            } else {
                $product['offer'] = null;
                $product['final_price'] = $product['price'];
            }

            return $product;
        }, $products);
    }
}
```

FASE 4: Crear los Controladores

1. Generar los Controladores

En esta fase vamos a crear los siguientes controladores, cada uno con una responsabilidad clara dentro de la arquitectura MVC de nuestra tienda online:

- **WelcomeController**: Este controlador gestionará la página de inicio (welcome) de la tienda online. Mostrará contenido destacado, información general y enlaces a las diferentes secciones principales de la tienda.
- **ProductController**: Se encargará de gestionar todo lo relacionado con los productos. Mostrará la página principal con el listado de productos, el detalle de un producto concreto y permitirá filtrar productos por categoría si es necesario.
- **CategoryController**: Este controlador gestionará las categorías de productos. Permitirá mostrar todos los productos de una categoría específica y navegar entre diferentes categorías.
- **OfferController**: Su función será mostrar la información de las ofertas disponibles y, si se desea, filtrar productos que tengan una oferta concreta.
- **CartController**: Se encargará de la gestión del carrito de la compra. Mostrará el estado actual del carrito, los productos añadidos, cantidades y permitirá ver el detalle de un carrito específico.

Cada uno de estos controladores nos ayudará a organizar la lógica de la aplicación, facilitando el mantenimiento y la escalabilidad del proyecto.

Ejecuta los siguientes comandos en la terminal:

```
sail artisan make:controller WelcomeController
sail artisan make:controller ProductController --resource
sail artisan make:controller CategoryController
sail artisan make:controller OfferController
sail artisan make:controller CartController
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ sail artisan make:controller
WelcomeController
Controller created successfully.

ggarrido@mi-equipo:~/development/laravel/MyShop$ sail artisan make:controller
ProductController --resource
Controller created successfully.

ggarrido@mi-equipo:~/development/laravel/MyShop$ sail artisan make:controller
CategoryController
Controller created successfully.

ggarrido@mi-equipo:~/development/laravel/MyShop$ sail artisan make:controller
OfferController
```

```
Controller created successfully.

ggarrido@mi-equipo:~/development/laravel/MyShop$ sail artisan make:controller
CartController
Controller created successfully.
```

2. Implementar WelcomeController

Este controlador maneja la página de bienvenida (`/`) que ya creamos en la práctica 1. Su función principal es cargar los datos necesarios para mostrar:

- **Categorías destacadas**: Las primeras 4 categorías de productos
- **Productos destacados**: Los primeros 3 productos con datos de ofertas para mostrar en la sección de destacados

Abre el archivo `app/Http/Controllers/WelcomeController.php` y reemplaza su contenido:

```php
<?php

namespace App\Http\Controllers;

use App\Traits\LoadsMockData;
use Illuminate\Http\Request;
use Illuminate\View\View;

class WelcomeController extends Controller
{
    use LoadsMockData;

    /**
     * Show the welcome page with featured content
     */
    public function index(): View
    {
        $products = $this->getProducts();
        $categories = $this->getCategories();

        // Enrich products with offer data before slicing
        $enrichedProducts = $this->enrichProductsWithOffers($products);

        // Get featured products (first 3 products for the featured section)
        $featuredProducts = array_slice($enrichedProducts, 0, 3, true);

        // Get featured categories (first 4 categories for the categories section)
        $featuredCategories = array_slice($categories, 0, 4, true);

        return view('welcome', compact('featuredProducts', 'featuredCategories'));
    }
}
```

3. Implementar ProductController (Resource)

El `ProductController` es un controlador de recurso en Laravel que gestiona todas las operaciones relacionadas con los productos de la tienda online. Al ser un controlador de recurso, implementa los métodos estándar del patrón CRUD (Crear, Leer, Actualizar, Eliminar).

Métodos implementados en `ProductController`

- **index()**: Muestra un listado de todos los productos. **Optimizado**: Carga solo productos y los enriquece con ofertas usando `enrichProductsWithOffers()`. No carga categorías ni carrito innecesarios.
- **onSale()**: Muestra solo los productos que tienen una oferta activa. Enriquece productos con ofertas y filtra donde `offer !== null`. **Este método se usa para el botón "🏷️ Ofertas Especiales".**
- **create()**: Muestra el formulario para crear un nuevo producto. En este ejemplo, simplemente redirige a la lista de productos con un mensaje simulado.
- **store(Request $request)**: Procesa el formulario de creación de un producto. Por ahora, solo redirige a la lista de productos sin guardar nada realmente.
- **show($id)**: Muestra el detalle de un producto específico. **Optimizado**: Carga solo el producto específico y su categoría. El producto se enriquece con datos de su oferta.
- **edit($id)**: Muestra el formulario para editar un producto existente. En este ejemplo, solo redirige a la lista de productos con un mensaje simulado.
- **update(Request $request, $id)**: Procesa el formulario de edición de un producto. Actualmente, solo redirige a la lista de productos sin modificar nada.
- **destroy($id)**: Elimina un producto. En este ejemplo, simplemente redirige a la lista de productos con un mensaje simulado.

> [!NOTE] Nota En esta fase, la mayoría de los métodos no tienen lógica real de persistencia, ya que los datos se gestionan mediante archivos mock y no se guardan cambios. Más adelante, cuando se conecte con una base de datos real, estos métodos implementarán la funcionalidad completa.

A continuación, se muestra la implementación completa del `ProductController` con todos los métodos mencionados, utilizando los datos mock para simular el comportamiento de la tienda.

Abre el archivo `app/Http/Controllers/ProductController.php` y reemplaza su contenido:

```php
<?php

namespace App\Http\Controllers;

use App\Traits\LoadsMockData;
use Illuminate\Http\Request;
use Illuminate\View\View;

class ProductController extends Controller
{
    use LoadsMockData;

    /**
     * Display a listing of the resource.
     */
```

```php
    public function index(): View
    {
        $products = $this->getProducts();

        // Enrich products with offer data and calculate final prices
        $enrichedProducts = $this->enrichProductsWithOffers($products);

        return view('products.index', ['products' => $enrichedProducts]);
    }

    /**
     * Display only products that have an active offer
     */
    public function onSale(): View
    {
        $products = $this->getProducts();
        $enrichedProducts = $this->enrichProductsWithOffers($products);

        // Filter only products with offers
        $productsOnSale = array_filter($enrichedProducts, function($product) {
            return $product['offer'] !== null;
        });

        return view('products.index', ['products' => $productsOnSale]);
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        // En una aplicación real, aquí se mostraría un formulario para crear un
nuevo producto.
        // En este ejemplo, simplemente redirigimos a la lista de productos.
        return redirect()->route('products.index')
            ->with('success', 'Formulario de creación de producto (simulado)');
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        // En una aplicación real, aquí se guardaría en la base de datos
        // Por ahora, solo redirigimos a la lista de productos
        return redirect()->route('products.index')
            ->with('success', 'Producto creado exitosamente');
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id): View
    {
```

```
        // Validate ID format
        if (!is_numeric($id) || $id < 1) {
            abort(404, 'ID de producto inválido');
        }

        $products = $this->getProducts();

        // Find product by ID
        $product = $products[$id] ?? null;

        if (!$product) {
            abort(404, 'Producto no encontrado');
        }

        // Enrich product with offer data
        $enrichedProducts = $this->enrichProductsWithOffers([$id => $product]);
        $product = $enrichedProducts[$id];

        // Get product category
        $categories = $this->getCategories();
        $category = $categories[$product['category_id']] ?? null;

        return view('products.show', compact('product', 'category'));
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        // En una aplicación real, aquí se obtendrían los datos del producto
correspondiente al id recibido,
        // así como las listas necesarias (por ejemplo, categorías, ofertas, etc.)
para mostrarlas en un formulario de edición.
        // En este ejemplo, simplemente redirigimos al detalle del producto.
        return redirect()->route('products.show', $id)
            ->with('success', 'Producto editado');
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, string $id)
    {
        // En una aplicación real, aquí se actualizaría en la base de datos
        // Por ahora, solo redirigimos al detalle del producto
        return redirect()->route('products.show', $id)
            ->with('success', 'Producto actualizado exitosamente');
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
```

```
    {
        // En una aplicación real, aquí se eliminaría de la base de datos
        // Por ahora, solo redirigimos a la lista de productos
        return redirect()->route('products.index')
            ->with('success', 'Producto eliminado exitosamente');
    }
}
```

4. Implementar CategoryController

El `CategoryController` es el encargado de gestionar todas las operaciones relacionadas con las categorías de productos en la tienda online. Su función principal es mostrar el listado de categorías y los productos asociados a cada una de ellas.

Métodos implementados en `CategoryController`

- **index()**: Este método muestra un listado de todas las categorías disponibles en la tienda. Es útil para que el usuario pueda navegar por las diferentes categorías.
- **show($id)**: Muestra todos los productos de una categoría específica. Si la categoría no existe, devuelve un error 404.

En resumen, el `CategoryController` organiza y facilita la navegación por categorías y sus productos. Esto mejora la experiencia del usuario al explorar el catálogo de la tienda online.

Abre el archivo `app/Http/Controllers/CategoryController.php` y reemplaza su contenido:

```php
<?php

namespace App\Http\Controllers;

use App\Traits\LoadsMockData;
use Illuminate\Http\Request;
use Illuminate\View\View;

class CategoryController extends Controller
{
    use LoadsMockData;

    /**
     * Show all categories
     */
    public function index(): View
    {
        $categories = $this->getCategories();

        return view('categories.index', ['categories' => $categories]);
    }

    /**
     * Show products from a specific category
```

```
     */
    public function show(string $id): View
    {
        // Validate ID format
        if (!is_numeric($id) || $id < 1) {
            abort(404, 'ID de categoría inválido');
        }

        $categories = $this->getCategories();

        // Find category by ID
        $category = $categories[$id] ?? null;

        if (!$category) {
            abort(404, 'Categoría no encontrada');
        }

        // Load and enrich products
        $products = $this->getProducts();

        // Filter products by category
        $categoryProducts = array_filter($products, function($product) use ($id) {
            return $product['category_id'] == $id;
        });

        $categoryProducts = $this->enrichProductsWithOffers($categoryProducts);

        return view('categories.show', compact('category', 'categoryProducts'));
    }
}
```

5. Implementar OfferController

El `OfferController` es el encargado de gestionar las operaciones relacionadas con las ofertas disponibles en la tienda online. Su función principal es mostrar el listado de todas las ofertas y permitir visualizar los productos asociados a una oferta concreta.

Métodos implementados en `OfferController`

- **index()**: Muestra un listado de todas las ofertas disponibles.
- **show($id)**: Muestra el detalle de una oferta específica y los productos que tienen esa oferta.

A continuación, se muestra la implementación completa del `OfferController` con los métodos mencionados, utilizando los datos mock para simular el comportamiento de la tienda.

Abre el archivo `app/Http/Controllers/OfferController.php` y reemplaza su contenido:

```
<?php

namespace App\Http\Controllers;
```

```php
use App\Traits\LoadsMockData;
use Illuminate\Http\Request;
use Illuminate\View\View;

class OfferController extends Controller
{
    use LoadsMockData;

    /**
     * Show all offers
     */
    public function index(): View
    {
        $offers = $this->getOffers();

        return view('offers.index', ['offers' => $offers]);
    }

    /**
     * Show products with a specific offer
     */
    public function show(string $id): View
    {
        // Validate ID format
        if (!is_numeric($id) || $id < 1) {
            abort(404, 'ID de oferta inválido');
        }

        $offers = $this->getOffers();

        // Find offer by ID
        $offer = $offers[$id] ?? null;

        if (!$offer) {
            abort(404, 'Oferta no encontrada');
        }

        // Load and enrich products
        $products = $this->getProducts();

        // Filter products by offer (un producto solo puede tener una oferta)
        $offerProducts = array_filter($products, function($product) use ($id) {
            return $product['offer_id'] == $id;
        });

        $offerProducts = $this->enrichProductsWithOffers($offerProducts);

        return view('offers.show', compact('offer', 'offerProducts'));
    }
}
```

6. Implementar CartController

El `CartController` es el encargado de gestionar todas las operaciones relacionadas con el carrito de la compra en la tienda online. Su función principal es mostrar el estado actual del carrito, permitiendo al usuario ver los productos añadidos, sus cantidades y detalles, así como visualizar la información de un producto específico dentro del carrito.

Métodos implementados en `CartController`

- **index()**: Este método muestra un resumen del carrito de la compra, incluyendo los productos añadidos, sus nombres y precios. Es útil para que el usuario pueda revisar el contenido de su carrito antes de finalizar la compra.
- **store(Request $request)**: Este método añade un nuevo producto al carrito. Valida que la cantidad sea un número entre 0 y 99. En una aplicación real, guardaría el producto en la sesión o base de datos del carrito.
- **update(Request $request, $id)**: Este método actualiza la cantidad de un producto existente en el carrito. Valida que la cantidad sea un número entre 0 y 99. Si la cantidad es 0, se eliminaría el producto del carrito.

En resumen, el `CartController` facilita la gestión del carrito de la compra, permitiendo añadir productos y actualizar cantidades con validación adecuada.

Abre el archivo `app/Http/Controllers/CartController.php` y reemplaza su contenido:

```php
<?php

namespace App\Http\Controllers;

use App\Traits\LoadsMockData;
use Illuminate\Http\Request;
use Illuminate\View\View;

class CartController extends Controller
{
    use LoadsMockData;

    /**
     * Show cart overview
     */
    public function index(): View
    {
        $cart = $this->getCart();
        $products = $this->getProducts();

        // Add product names to cart data
        $cartWithProducts = [];
        foreach ($cart as $item) {
            $product = $products[$item['product_id']] ?? null;
            $cartWithProducts[] = array_merge($item, [
```

```
                'name' => $product ? $product['name'] : 'Producto no encontrado',
                'price' => $product ? $product['price'] : 0
            ]);
        }

        return view('cart.index', [
            'cartItems' => $cartWithProducts
        ]);
    }

    /**
     * Store a newly created cart item
     */
    public function store(Request $request)
    {
        // En una aplicación real, aquí se guardaría el producto en el carrito
        // Por ahora, solo redirigimos al carrito con un mensaje
        return redirect()->route('cart.index')
            ->with('success', 'Producto añadido al carrito exitosamente');
    }

    /**
     * Update the specified cart item
     */
    public function update(Request $request, string $id)
    {
        // En una aplicación real, aquí se actualizaría la cantidad del producto
en el carrito
        // Por ahora, solo redirigimos al carrito con un mensaje
        return redirect()->route('cart.index')
            ->with('success', 'Cantidad actualizada exitosamente');
    }
}
```

[!NOTE] Controladores Resource vs Individuales

En esta práctica mostramos ambos enfoques:

- **ProductController**: Es un controlador **resource** con todos los métodos CRUD (`index`, `create`, `store`, `show`, `edit`, `update`, `destroy`)
- **CategoryController, OfferController, CartController**: Son controladores **individuales** con métodos específicos para funcionalidades concretas
- **WelcomeController**: Es un controlador **individual** para la página de bienvenida

¿Cuándo usar cada uno?

- **Resource**: Cuando necesitas operaciones CRUD completas (como productos)
- **Individual**: Cuando solo necesitas funcionalidades específicas (como mostrar categorías o ofertas)

En el futuro algunos controladores cambiarán completando los métodos para convertirse en un `controlador resource`.

FASE 5: Definir las Rutas Públicas

Ahora que tenemos los controladores creados, podemos definir las rutas que conectarán las URLs con estos controladores.

1. Explicación de las Rutas

En esta práctica, vamos a definir varias rutas públicas en el archivo `routes/web.php` para cubrir las necesidades básicas de una tienda online. Cada ruta conecta una URL con un controlador específico que se encargará de la lógica correspondiente.

Las rutas que vamos a crear son:

- **Ruta principal (`/`)**: Muestra la página de inicio con productos y categorías destacados
- **Lista de productos (`/products`)**: Muestra todos los productos disponibles
- **Productos en oferta (`/products-on-sale`)**: Muestra solo los productos que tienen oferta activa
- **Detalle de producto (`/products/{id}`)**: Permite ver la información completa de un producto concreto
- **Lista de categorías (`/categories`)**: Muestra todas las categorías disponibles
- **Detalle de categoría (`/categories/{id}`)**: Muestra todos los productos de una categoría específica
- **Lista de ofertas (`/offers`)**: Muestra todas las ofertas disponibles
- **Detalle de oferta (`/offers/{id}`)**: Muestra información sobre una oferta específica y sus productos
- **Carrito de la compra (`/cart`)**: Permite ver y gestionar el carrito
- **Contacto (`/contact`)**: Página de contacto

2. Configurar las Rutas en `routes/web.php`

Abre el archivo `routes/web.php` y reemplaza el contenido por las siguientes rutas:

```php
<?php

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\WelcomeController;
use App\Http\Controllers\ProductController;
use App\Http\Controllers\CategoryController;
use App\Http\Controllers\OfferController;
use App\Http\Controllers\CartController;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
*/

// Welcome page - shows home page with featured content
Route::get('/', [WelcomeController::class, 'index'])->name('welcome');

// Contact page
Route::get('/contact', function () {
    return view('contact');
})->name('contact');
```

```
// Category routes
Route::get('/categories', [CategoryController::class, 'index'])-
>name('categories.index');
Route::get('/categories/{id}', [CategoryController::class, 'show'])-
>name('categories.show');

// Cart routes
Route::get('/cart', [CartController::class, 'index'])->name('cart.index');
Route::post('/cart', [CartController::class, 'store'])->name('cart.store');
Route::put('/cart/{id}', [CartController::class, 'update'])
    ->where('id', '[0-9]+')
    ->name('cart.update');

// Special route for products on sale (must be BEFORE resource routes to avoid
conflicts)
Route::get('/products-on-sale', [ProductController::class, 'onSale'])-
>name('products.on-sale');

// Resource routes - Example with products (creates all CRUD routes automatically)
Route::resource('products', ProductController::class);

// OfferController: only index y show
Route::resource('offers', OfferController::class)->only(['index', 'show']);
```

3. Verificar las Rutas

Puedes verificar que todas las rutas se han creado correctamente ejecutando:

```
sail artisan route:list
```

```
  GET|HEAD  / ..................................... welcome ›
WelcomeController@index
  GET|HEAD  cart ................................... cart.index ›
CartController@index
  POST      cart ................................... cart.store ›
CartController@store
  PUT       cart/{id} .............................. cart.update ›
CartController@update
  GET|HEAD  categories ............................. categories.index ›
CategoryController@index
  GET|HEAD  categories/{id} ........................ categories.show ›
CategoryController@show
  GET|HEAD  contact ................................ contact
  GET|HEAD  offers ................................. offers.index ›
OfferController@index
  GET|HEAD  offers/{offer} ......................... offers.show ›
OfferController@show
  GET|HEAD  products ............................... products.index ›
```

```
ProductController@index
  POST      products ..................................... products.store ›
ProductController@store
  GET|HEAD  products-on-sale ............................ products.on-sale ›
ProductController@onSale
  GET|HEAD  products/create ............................. products.create ›
ProductController@create
  GET|HEAD  products/{product} .......................... products.show ›
ProductController@show
  PUT|PATCH products/{product} .......................... products.update ›
ProductController@update
  DELETE    products/{product} .......................... products.destroy ›
ProductController@destroy
  GET|HEAD  products/{product}/edit ..................... products.edit ›
ProductController@edit
```

Este comando mostrará una tabla con todas las rutas disponibles, sus métodos HTTP, URIs y los controladores/acciones asociados.

> [!CAUTION] Captura de pantalla requerida Realiza una captura de pantalla de tu terminal mostrando:
>
> La salida del comando `sail artisan route:list` similar a la del ejemplo sin las rutas de `telescope`
>
> La captura debe incluir tu prompt `usuario@equipo:~/ruta/proyecto$` visible.

FASE 6: Crear el Sistema de Layouts, Partials y Componentes

1. Crear la Estructura de Directorios

Vamos a implementar un **sistema híbrido moderno** que combina lo mejor de tres enfoques complementarios:

- **Layout maestro**: Define la estructura general de la página (HTML, head, body) y permite incluir contenido dinámico mediante secciones (`@yield`, `@section`). Es como el "esqueleto" de todas las páginas.
- **Partials**: Son fragmentos HTML simples para elementos fijos como el head, header, footer, navegación, etc. Útiles para elementos que no necesitan lógica compleja.
- **Componentes Blade**: Permiten encapsular bloques de interfaz reutilizables y parametrizables (tarjetas de producto, alertas, botones, etc.). Son como "piezas de LEGO" que se pueden combinar y personalizar.

Crea la estructura de directorios:

```
mkdir -p resources/views/layouts
mkdir -p resources/views/partials
mkdir -p resources/views/components
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p resources/views/layouts
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p resources/views/partials
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p
resources/views/components
ggarrido@mi-equipo:~/development/laravel/MyShop$ ls -la resources/views/
total 20
drwxrwxr-x 5 ggarrido ggarrido 4096 ene 15 10:40 .
drwxrwxr-x 5 ggarrido ggarrido 4096 ene 15 10:40 ..
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:40 components
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:40 layouts
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:40 partials
-rw-rw-r-- 1 ggarrido ggarrido 1234 ene 15 10:40 welcome.blade.php
```

2. Crear los Partials (Elementos Fijos)

Primero creamos los partials para los elementos que se mantienen fijos en todas las páginas. Los partials son fragmentos HTML simples que se incluyen en el layout maestro.

2.1. Head Partial

Este partial `head.blade.php` contendrá las etiquetas meta principales, el título de la página (usando `@yield('title')` para que cada vista pueda cambiarlo), la inclusión de TailwindCSS y la configuración de colores personalizados, así como un stack para estilos adicionales (`@stack('styles')`). Es decir, aquí se centralizan los elementos del `<head>` que serán comunes a todas las páginas.

Crea el archivo `resources/views/partials/head.blade.php`:

```
touch resources/views/partials/head.blade.php
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@yield('title', 'Mi Tienda Online')</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <script>
        tailwind.config = {
            theme: {
                extend: {
                    colors: {
                        primary: {
                            500: '#3b82f6',
                            600: '#2563eb',
                            700: '#1d4ed8',
                        }
                    }
                }
            }
        }
```

```
    </script>
    @stack('styles')
```

2.2. Header Partial

Este partial `header.blade.php` contendrá el encabezado principal del sitio, incluyendo el logo, la navegación principal (que se incluirá mediante el partial `navigation.blade.php`) y el acceso al carrito de compras. Su objetivo es proporcionar una barra superior consistente en todas las páginas, facilitando la navegación y el acceso rápido al carrito.

Crea el archivo `resources/views/partials/header.blade.php`:

```
touch resources/views/partials/header.blade.php
```

```
<!-- Header con navegación -->
<header class="bg-white shadow-lg relative">
        <div class="container mx-auto px-6 py-4">
            <div class="flex items-center justify-between">
                <!-- Logo -->
                <div class="flex items-center space-x-4">
                <a href="{{ route('welcome') }}" class="text-2xl font-bold text-
primary-600">
                        🛍️ Mi Tienda
                    </a>
                </div>

                <!-- Navegación usando partial -->
                @include('partials.navigation')

                <!-- Carrito -->
                <div class="flex items-center space-x-4">
                <a href="{{ route('cart.index') }}"
                   class="text-gray-700 hover:text-primary-600 transition">
                        🛒 Carrito (0)
                </a>
                </div>
            </div>
        </div>
    </header>
```

2.3. Navigation Partial

Este partial `navigation.blade.php` contendrá la barra de navegación principal del sitio. Incluirá enlaces a las secciones más importantes como Inicio, Productos, Categorías, Ofertas y Contacto. Además, resaltará el enlace activo según la ruta actual, facilitando la navegación y mejorando la experiencia del usuario.

Crea el archivo `resources/views/partials/navigation.blade.php`:

```
touch resources/views/partials/navigation.blade.php
```

```
<nav class="hidden md:flex space-x-8">
    <a href="{{ route('welcome') }}"
       class="text-gray-700 hover:text-primary-600 transition {{ request()-
>routeIs('welcome') ? 'text-primary-600 font-semibold' : '' }}">
        Inicio
    </a>
    <a href="{{ route('products.index') }}"
       class="text-gray-700 hover:text-primary-600 transition {{ request()-
>routeIs('products.*') ? 'text-primary-600 font-semibold' : '' }}">
        Productos
    </a>
    <a href="{{ route('categories.index') }}"
       class="text-gray-700 hover:text-primary-600 transition {{ request()-
>routeIs('categories.*') ? 'text-primary-600 font-semibold' : '' }}">
        Categorías
    </a>
    <a href="{{ route('offers.index') }}"
       class="text-gray-700 hover:text-primary-600 transition {{ request()-
>routeIs('offers.*') ? 'text-primary-600 font-semibold' : '' }}">
        Ofertas
    </a>
    <a href="{{ route('contact') }}"
       class="text-gray-700 hover:text-primary-600 transition {{ request()-
>routeIs('contact') ? 'text-primary-600 font-semibold' : '' }}">
        Contacto
    </a>
</nav>
```

2.4. Footer Partial

Este partial contendrá el pie de página (footer) del sitio web. Incluirá información sobre la tienda, enlaces rápidos a las secciones principales (Inicio, Productos, Categorías, Contacto) y datos de contacto. Además, mostrará un mensaje de derechos reservados en la parte inferior.

Crea el archivo `resources/views/partials/footer.blade.php`:

```
touch resources/views/partials/footer.blade.php
```

```
    <!-- Footer -->
    <footer class="bg-gray-800 text-white py-8 mt-12">
        <div class="container mx-auto px-6">
            <div class="grid grid-cols-1 md:grid-cols-3 gap-8">
                <div>
```

```
                    <h5 class="text-xl font-bold mb-4">🛍️ Mi Tienda</h5>
                    <p class="text-gray-400">
                        Tu tienda de confianza para encontrar los mejores
productos.
                    </p>
                </div>
                <div>
                    <h6 class="font-bold mb-4">Enlaces Rápidos</h6>
                    <ul class="space-y-2 text-gray-400">
                    <li><a href="{{ route('welcome') }}" class="hover:text-white
transition">Inicio</a></li>
                    <li><a href="{{ route('products.index') }}" class="hover:text-
white transition">Productos</a></li>
                    <li><a href="{{ route('categories.index') }}"
class="hover:text-white transition">Categorías</a></li>
                    <li><a href="#" class="hover:text-white
transition">Contacto</a></li>
                    </ul>
                </div>
                <div>
                    <h6 class="font-bold mb-4">Contacto</h6>
                    <ul class="space-y-2 text-gray-400">
                        <li>📞 Teléfono de contacto</li>
                        <li>📧 Email de contacto</li>
                        <li>🕒 Horario de atención</li>
                    </ul>
                </div>
            </div>
            <div class="border-t border-gray-700 mt-8 pt-8 text-center text-gray-
400">
                <p>&copy; 2025 Mi Tienda. Todos los derechos reservados.</p>
            </div>
        </div>
    </footer>
```

3. Crear el Layout Maestro

El layout maestro es la base de todas las páginas. Define la estructura HTML común y utiliza los partials para los elementos fijos.

Crea el archivo `resources/views/layouts/app.blade.php`:

```
touch resources/views/layouts/app.blade.php
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    @include('partials.head')
</head>
```

```
<body class="bg-gray-50">
    <!-- Header usando partial -->
    @include('partials.header')

    <!-- Contenido principal -->
    <main class="min-h-screen">
        @yield('content')
    </main>

    <!-- Footer usando partial -->
    @include('partials.footer')

    @stack('scripts')
</body>
</html>
```

4. Crear Componentes Blade para Elementos Reutilizables

Los componentes Blade son la evolución moderna de los partials. Permiten crear elementos reutilizables que encapsulan su presentación y su lógica.

Estos pueden reutilizarse gracias al uso de **Props**, que son datos que se pasan al componente dinámicamente (como parámetros de una función).

4.1. Product Card Component

Este componente `ProductCard` es el componente más importante de la tienda. Contendrá la presentación de un producto individual en forma de tarjeta y se **adaptará visualmente** según si el producto tiene oferta o no.

Crea el componente `ProductCard`:

```
sail artisan make:component ProductCard
```

Esto creará dos archivos:

- `app/View/Components/ProductCard.php` (lógica del componente)
- `resources/views/components/product-card.blade.php` (vista del componente)

Archivo `app/View/Components/ProductCard.php`:

En este archivo se define la lógica del componente `ProductCard`. Aquí se especifica qué propiedades (props) debe recibir el componente para funcionar correctamente. Por ejemplo, espera recibir un array con los datos del producto (`product`). Además, puede recibir una cadena (`class`) para personalizar las clases CSS y así adaptar el estilo del componente según sea necesario.

```
<?php

namespace App\View\Components;
```

```php
use Closure;
use Illuminate\Contracts\View\View;
use Illuminate\View\Component;

class ProductCard extends Component
{
    /**
     * Create a new component instance.
     */
    public function __construct(
        public array $product,
        public string $class = ''
    ) {}

    /**
     * Get the view / contents that represent the component.
     */
    public function render(): View|Closure|string
    {
        return view('components.product-card');
    }
}
```

Archivo `resources/views/components/product-card.blade.php`:

Este archivo `resources/views/components/product-card.blade.php` contendrá la vista del componente Blade `ProductCard`. Aquí se define la estructura HTML y el diseño visual de la tarjeta de producto, utilizando las propiedades (`$product`, `$class`) que recibe el componente.

```blade
<div class="bg-white rounded-lg shadow-lg overflow-hidden product-card {{ $class
}} relative {{ $product['offer'] !== null ? 'ring-2 ring-orange-400' : '' }}">
    <!-- Badge de oferta destacado (esquina superior derecha) -->
    @if($product['offer'] !== null)
        <div class="absolute top-0 right-0 bg-gradient-to-r from-orange-500 to-
red-500 text-white px-4 py-2 rounded-bl-lg font-bold shadow-lg z-10">
            <span class="text-lg">
                -{{ $product['offer']['discount_percentage'] }}%
            </span>
        </div>
    @endif

    <div class="h-48 bg-gray-200 flex items-center justify-center {{
$product['offer'] !== null ? 'bg-gradient-to-br from-orange-50 to-red-50' : ''
}}">
        <span class="text-4xl">📦</span>
    </div>
    <div class="p-6">
        <h4 class="text-xl font-bold mb-2 text-gray-900">{{ $product['name'] }}
</h4>
        <p class="text-gray-600 mb-4">{{ $product['description'] }}</p>
```

```
        <!-- Badge de oferta adicional (nombre de la oferta) -->
        @if($product['offer'] !== null)
            <div class="mb-4">
                <span class="inline-block bg-orange-100 text-orange-800 text-xs
px-3 py-1 rounded-full font-semibold">
                    🏷️ {{ $product['offer']['name'] }}
                    </span>
            </div>
        @endif

        <div class="flex items-center justify-between flex-wrap gap-2">
            <div class="flex flex-col">
                @if($product['offer'] !== null)
                    <span class="text-sm text-gray-400 line-through">€{{
number_format($product['price'], 2) }}</span>
                    <span class="text-2xl font-bold text-orange-600">€{{
number_format($product['final_price'], 2) }}</span>
                @else
                    <span class="text-2xl font-bold text-primary-600">€{{
number_format($product['price'], 2) }}</span>
                @endif
            </div>
            <a href="{{ route('products.show', $product['id']) }}"
               class="bg-primary-600 text-white px-4 py-2 rounded-lg hover:bg-
primary-700 transition">
                Ver Detalles
            </a>
        </div>
    </div>
</div>
```

4.2. Category Card Component

Este componente `CategoryCard` contendrá la presentación de una categoría individual en forma de tarjeta. Mostrará información relevante de la categoría, como el nombre, la descripción, un icono representativo, y permitirá personalizar su apariencia mediante la prop `class`.

Crea el componente `CategoryCard`:

```
sail artisan make:component CategoryCard
```

Archivo `app/View/Components/CategoryCard.php`:

En este archivo se define la lógica del componente `CategoryCard` que tendrá un comportamiento similar al de `producto` visto previamente.

```
<?php
```

```php
namespace App\View\Components;

use Closure;
use Illuminate\Contracts\View\View;
use Illuminate\View\Component;

class CategoryCard extends Component
{
    /**
     * Create a new component instance.
     */
    public function __construct(
        public array $category,
        public string $class = ''
    ) {}

    /**
     * Get the view / contents that represent the component.
     */
    public function render(): View|Closure|string
    {
        return view('components.category-card');
    }
}
```

Archivo `resources/views/components/category-card.blade.php`:

Este archivo `resources/views/components/category-card.blade.php` contendrá la vista del componente Blade `CategoryCard`.

```blade
<div class="bg-white rounded-lg shadow-lg p-6 product-card cursor-pointer {{
$class }}">
    <div class="text-4xl text-primary-500 mb-4">📦</div>
    <h4 class="text-xl font-bold mb-2 text-gray-900">{{ $category['name'] }}</h4>
    <p class="text-gray-600 mb-4">{{ $category['description'] }}</p>
    <a href="{{ route('categories.show', $category['id']) }}"
       class="text-primary-600 font-semibold hover:text-primary-700 transition">
        Ver Productos →
    </a>
</div>
```

FASE 7: Refactorizar la Vista Welcome

Ahora que tenemos nuestro sistema de layouts, partials y componentes, es el momento de **refactorizar la vista `welcome.blade.php`** para que aproveche toda esta nueva arquitectura. Esto nos permitirá:

- Eliminar el código HTML duplicado (header, footer, etc.)
- Usar los componentes reutilizables que hemos creado
- Mantener un código más limpio y mantenible
- Integrar los datos dinámicos que vienen del controlador

1. Actualizar la Vista Welcome

Vamos a reemplazar el contenido completo del archivo `welcome.blade.php` para que use el layout maestro, los componentes y los datos dinámicos del controlador.

Abre el archivo `resources/views/welcome.blade.php` y reemplaza su contenido completo:

```
@extends('layouts.app')

@section('title', 'Bienvenido - Mi Tienda')

@push('styles')
    <style>
        .hero-gradient {
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
        }
    </style>
@endpush

@section('content')
    <!-- Hero Section -->
    <section class="hero-gradient text-white py-20">
        <div class="container mx-auto px-6 text-center">
            <h2 class="text-4xl md:text-6xl font-extrabold leading-tight mb-6">
                Bienvenido a Mi Tienda
            </h2>
            <p class="text-xl md:text-2xl text-blue-100 mb-8 max-w-3xl mx-auto">
                Descubre una amplia variedad de productos de calidad.
                Encuentra lo que buscas al mejor precio.
            </p>
            <div class="flex flex-wrap justify-center gap-4">
                <a href="{{ route('products.index') }}"
                   class="bg-white text-primary-600 font-bold py-4 px-8 rounded-
full hover:bg-gray-100 transition duration-300 ease-in-out transform hover:scale-
105">
                    Ver Productos
                </a>
                <a href="{{ route('products.on-sale') }}"
                   class="border-2 border-white text-white font-bold py-4 px-8
rounded-full hover:bg-white hover:text-primary-600 transition duration-300 ease-
in-out">
                    🏷️ Ofertas Especiales
                </a>
            </div>
        </div>
    </section>

    <!-- Categorías Destacadas -->
    <section class="py-16">
        <div class="container mx-auto px-6">
            <h3 class="text-3xl font-bold mb-12 text-center text-gray-900">
                Nuestras Categorías
```

```
            </h3>
            <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-8">
                @forelse($featuredCategories as $category)
                    <x-category-card :category="$category" />
                @empty
                    <div class="col-span-full text-center py-12">
                        <p class="text-gray-500 text-lg">No hay categorías
disponibles.</p>
                    </div>
                @endforelse
            </div>
        </div>
    </section>

    <!-- Productos Destacados -->
    <section class="py-16 bg-gray-100">
        <div class="container mx-auto px-6">
            <h3 class="text-3xl font-bold mb-12 text-center text-gray-900">
                Productos Destacados
            </h3>
            <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-8">
                @forelse($featuredProducts as $product)
                    <x-product-card :product="$product" />
                @empty
                    <div class="col-span-full text-center py-12">
                        <p class="text-gray-500 text-lg">No hay productos
destacados.</p>
                    </div>
                @endforelse
            </div>
        </div>
    </section>
@endsection
```

2. Verificar el Funcionamiento

Una vez refactorizada la vista, verifica que todo funciona correctamente:

1. Accede a `http://localhost` en tu navegador
2. Deberías ver la página de inicio con:
    - Header y navegación funcional
    - Hero section con botones que funcionan
    - Las 4 primeras categorías de tus datos mock
    - Los 3 primeros productos de tus datos mock
    - Productos con oferta destacados visualmente
    - Footer con enlaces
3. Haz clic en los enlaces de navegación para verificar que funcionan

> [!NOTE] Nota importante Si tu vista `welcome.blade.php` tenía personalizaciones adicionales (colores, iconos, textos específicos de tu temática), asegúrate de mantenerlas en la versión refactorizada. La estructura debe ser la misma, pero el contenido debe reflejar tu tienda personalizada.

FASE 8: Crear las Vistas del Catálogo

Ahora que tenemos nuestro sistema de layouts y componentes, y hemos refactorizado la vista principal, vamos a crear las vistas específicas para cada funcionalidad de la tienda. Cada vista se extiende del layout maestro y utiliza los componentes que hemos creado.

1. Crear Directorios de Vistas

Objetivo: Organizar las vistas por funcionalidad para mantener el código limpio y fácil de mantener. Cada directorio corresponde a una entidad específica.

Crea los directorios necesarios:

```
mkdir -p resources/views/products
mkdir -p resources/views/categories
mkdir -p resources/views/offers
mkdir -p resources/views/cart
```

```
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p resources/views/products
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p
resources/views/categories
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p resources/views/offers
ggarrido@mi-equipo:~/development/laravel/MyShop$ mkdir -p resources/views/cart
ggarrido@mi-equipo:~/development/laravel/MyShop$ ls -la resources/views/
total 20
drwxrwxr-x 7 ggarrido ggarrido 4096 ene 15 10:45 .
drwxrwxr-x 7 ggarrido ggarrido 4096 ene 15 10:45 ..
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 cart
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 categories
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 components
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 layouts
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 offers
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 partials
drwxrwxr-x 2 ggarrido ggarrido 4096 ene 15 10:45 products
-rw-rw-r-- 1 ggarrido ggarrido 1234 ene 15 10:40 welcome.blade.php
```

2. Vista de Lista de Productos

Vamos a crear la vista principal del catálogo, encargada de mostrar todos los productos disponibles en la tienda. Esta vista aprovecha el sistema de layouts y componentes para mantener una estructura coherente y un código reutilizable y limpio.

¿Cómo funciona esta vista?

1. **Extiende el layout maestro**: Utiliza `@extends('layouts.app')` para heredar la estructura base de la aplicación, incluyendo el encabezado, pie de página y los estilos globales. Así, todas las páginas mantienen una apariencia uniforme.

2. **Define un título personalizado**: Mediante `@section('title', 'Todos los Productos - Mi Tienda')`, cada página puede establecer su propio título, mejorando la experiencia de usuario y el SEO.
3. **Permite añadir estilos propios**: Utiliza la directiva `@push('styles')` para incluir CSS específico de esta página, sin afectar a otras vistas.
4. **Muestra datos dinámicos**: Recibe la lista de productos desde el controlador y la recorre para mostrar cada uno. Así, el contenido se adapta automáticamente a los datos disponibles.
5. **Utiliza componentes Blade**: Cada producto se representa mediante el componente `<x-product-card>`, lo que facilita la reutilización y el mantenimiento del código, además de asegurar una presentación consistente.
6. **Gestiona el caso de lista vacía**: Si no hay productos para mostrar, la vista presenta un mensaje amigable al usuario, evitando espacios vacíos o confusión.

Crea el archivo `resources/views/products/index.blade.php`:

```
touch resources/views/products/index.blade.php
```

```
@extends('layouts.app')

@section('title', 'Todos los Productos - Mi Tienda')

@push('styles')
    <style>
        .product-grid {
            display: grid;
            grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
            gap: 2rem;
        }
    </style>
@endpush

@section('content')
    <div class="container mx-auto px-6 py-8">
    <div class="mb-8">
        <h1 class="text-3xl font-bold text-gray-900 mb-4">Todos los Productos</h1>
        <p class="text-gray-600">Descubre nuestra amplia gama de productos de
calidad.</p>
    </div>

        <div class="product-grid">
        @forelse($products as $product)
                <x-product-card :product="$product" />
        @empty
            <div class="col-span-full text-center py-12">
                    <p class="text-gray-500 text-lg">No hay productos disponibles.
</p>
            </div>
        @endforelse
    </div>
```

```
    </div>
@endsection
```

3. Vista de Detalle de Producto

Esta vista está diseñada para mostrar de manera completa y atractiva toda la información relevante de un producto específico. Permite al usuario conocer en detalle las características del producto, visualizar las ofertas disponibles y realizar acciones relacionadas con la compra.

Crea el archivo `resources/views/products/show.blade.php`:

```
touch resources/views/products/show.blade.php
```

```
@extends('layouts.app')

@section('title', $product['name'] . ' - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
    <div class="grid grid-cols-1 lg:grid-cols-2 gap-8">
        <!-- Imagen del Producto -->
            <div class="bg-white rounded-lg shadow-lg p-6">
                <div class="h-96 bg-gray-200 flex items-center justify-center">
                    <span class="text-8xl">📦</span>
            </div>
        </div>

        <!-- Información del Producto -->
            <div class="bg-white rounded-lg shadow-lg p-6">
            <h1 class="text-3xl font-bold text-gray-900 mb-4">{{ $product['name']
}}</h1>
                <p class="text-gray-600 mb-6">{{ $product['description'] }}</p>

            <!-- Precio -->
            <div class="mb-6">
                @if($product['offer'] !== null)
                    <div class="flex items-baseline gap-3">
                        <span class="text-2xl text-gray-400 line-through">€{{
number_format($product['price'], 2) }}</span>
                        <span class="text-4xl font-bold text-orange-600">€{{
number_format($product['final_price'], 2) }}</span>
                    </div>
                    <p class="text-sm text-orange-600 mt-2">
                        ¡Ahorra €{{ number_format($product['price'] -
$product['final_price'], 2) }}!
                    </p>
                @else
                    <span class="text-4xl font-bold text-primary-600">€{{
number_format($product['price'], 2) }}</span>
```

```
                @endif
            </div>

                <!-- Categoría -->
                @if(isset($category))
            <div class="mb-6">
                        <span class="text-sm text-gray-500">Categoría:</span>
                        <a href="{{ route('categories.show', $category['id']) }}"
                           class="ml-2 bg-primary-100 text-primary-800 px-3 py-1
rounded-full text-sm hover:bg-primary-200 transition">
                            {{ $category['name'] }}
                        </a>
            </div>
                @endif

                <!-- Oferta -->
                @if($product['offer'] !== null)
            <div class="mb-6">
                        <span class="text-sm text-gray-500">Oferta activa:</span>
                        <div class="mt-2">
                                    <span class="inline-block bg-orange-100 text-
orange-800 text-sm px-3 py-1 rounded-full">
                                        🏷️ {{ $product['offer']['name'] }} (-{{
$product['offer']['discount_percentage'] }}%)
                        </span>
                </div>
            </div>
                        @endif

                <!-- Botones de Acción -->
            <div class="flex space-x-4">
                    <a href="{{ route('cart.store') }}"
                       class="bg-primary-600 text-white px-6 py-3 rounded-lg
hover:bg-primary-700 transition">
                        🛒 Añadir al Carrito
                    </a>
                    <a href="{{ route('products.index') }}"
                       class="border border-primary-600 text-primary-600 px-6 py-3
rounded-lg hover:bg-primary-50 transition">
                        ← Volver a Productos
                    </a>
            </div>
        </div>
    </div>
    </div>
@endsection
```

4. Vista de Lista de Categorías

Esta vista permite a los usuarios descubrir y explorar todas las categorías disponibles en la tienda, facilitando la navegación y el acceso a los productos organizados por temática o tipo.

Cada categoría se representa mediante el componente `CategoryCard`, lo que garantiza coherencia visual y facilita el mantenimiento del código.

Crea el archivo `resources/views/categories/index.blade.php`:

```
touch resources/views/categories/index.blade.php
```

```
@extends('layouts.app')

@section('title', 'Categorías - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
        <div class="mb-8">
            <h1 class="text-3xl font-bold text-gray-900 mb-4">Nuestras
Categorías</h1>
            <p class="text-gray-600">Explora nuestros productos por categoría.</p>
        </div>

        <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
            @forelse($categories as $category)
                <x-category-card :category="$category" />
            @empty
                <div class="col-span-full text-center py-12">
                    <p class="text-gray-500 text-lg">No hay categorías
disponibles.</p>
                </div>
            @endforelse
        </div>
    </div>
@endsection
```

5. Vista de Productos por Categoría

Esta vista está diseñada para mostrar todos los productos que pertenecen a una categoría específica, facilitando que los usuarios puedan explorar el catálogo de la tienda de manera temática o por tipo de producto.

Cada producto se representa mediante el componente `ProductCard`, lo que garantiza coherencia visual, facilita el mantenimiento y promueve la reutilización del código.

Crea el archivo `resources/views/categories/show.blade.php`:

```
touch resources/views/categories/show.blade.php
```

```
@extends('layouts.app')

@section('title', $category['name'] . ' - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
    <div class="mb-8">
        <h1 class="text-3xl font-bold text-gray-900 mb-4">{{ $category['name'] }}
</h1>
            <p class="text-gray-600 mb-4">{{ $category['description'] }}</p>
            <a href="{{ route('categories.index') }}"
               class="text-primary-600 hover:text-primary-700 transition">
                ← Volver a Categorías
            </a>
    </div>

        @if(!empty($categoryProducts))
        <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
            @foreach($categoryProducts as $product)
                    <x-product-card :product="$product" />
                            @endforeach
                        </div>
                                @else
            <div class="text-center py-12">
                <p class="text-gray-500 text-lg">No hay productos en esta
categoría.</p>
                            </div>
                        @endif
    </div>
@endsection
```

6. Vista de Lista de Ofertas

Esta vista está diseñada para mostrar todas las ofertas especiales activas en la tienda, permitiendo a los usuarios identificar rápidamente descuentos y promociones vigentes.

Crea el archivo `resources/views/offers/index.blade.php`:

```
touch resources/views/offers/index.blade.php
```

```
@extends('layouts.app')

@section('title', 'Ofertas - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
        <div class="mb-8">
            <h1 class="text-3xl font-bold text-gray-900 mb-4">Ofertas
```

```
Especiales</h1>
            <p class="text-gray-600">Descubre nuestras mejores ofertas y
descuentos.</p>
        </div>

        <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
            @forelse($offers as $offer)
                <div class="bg-white rounded-lg shadow-lg p-6 border-l-4 border-
orange-500">
                    <h3 class="text-xl font-bold text-gray-900 mb-2">{{
$offer['name'] }}</h3>
                    <p class="text-gray-600 mb-4">{{ $offer['description'] }}</p>
                    <div class="text-2xl font-bold text-orange-600 mb-4">
                        {{ $offer['discount_percentage'] }}% de descuento
                    </div>
                    <a href="{{ route('offers.show', $offer['id']) }}"
                       class="bg-orange-600 text-white px-4 py-2 rounded-lg
hover:bg-orange-700 transition">
                        Ver Productos
                            </a>
                        </div>
            @empty
                <div class="col-span-full text-center py-12">
                    <p class="text-gray-500 text-lg">No hay ofertas disponibles.
</p>
                    </div>
            @endforelse
                </div>
    </div>
@endsection
```

7. Vista de Detalle de Oferta

Esta vista muestra el detalle de una oferta específica y todos los productos que tienen aplicada esa oferta. Es útil para que los usuarios puedan ver de un vistazo todos los productos con descuento de una promoción concreta.

Crea el archivo `resources/views/offers/show.blade.php`:

```
touch resources/views/offers/show.blade.php
```

```
@extends('layouts.app')

@section('title', $offer['name'] . ' - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
        <!-- Header de la Oferta -->
        <div class="bg-gradient-to-r from-orange-500 to-red-500 rounded-lg shadow-
```

```
lg p-8 mb-8 text-white">
            <div class="flex items-center justify-between">
                <div>
                    <h1 class="text-4xl font-bold mb-2">{{ $offer['name'] }}</h1>
                    <p class="text-xl">{{ $offer['description'] }}</p>
                </div>
                <div class="bg-white text-orange-600 rounded-full w-32 h-32 flex
items-center justify-center">
                    <div class="text-center">
                        <div class="text-4xl font-bold">{{
$offer['discount_percentage'] }}%</div>
                        <div class="text-sm">OFF</div>
                    </div>
                </div>
            </div>
        </div>

        <!-- Productos con esta oferta -->
        <div class="mb-8">
            <h2 class="text-2xl font-bold text-gray-900 mb-6">Productos en
Oferta</h2>

            @if(!empty($offerProducts))
                <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
                    @foreach($offerProducts as $product)
                        <x-product-card :product="$product" />
                    @endforeach
                </div>
            @else
                <div class="text-center py-12 bg-gray-100 rounded-lg">
                    <p class="text-gray-500 text-lg">No hay productos con esta
oferta actualmente.</p>
                </div>
            @endif
        </div>

        <!-- Botón volver -->
        <div class="mt-8">
            <a href="{{ route('offers.index') }}"
               class="inline-block bg-gray-600 text-white px-6 py-3 rounded-lg
hover:bg-gray-700 transition">
                ← Volver a Ofertas
            </a>
        </div>
    </div>
@endsection
```

8. Vista del Carrito

Esta vista está diseñada para ofrecer al usuario una experiencia completa de gestión del carrito de compras. Permite visualizar de manera clara todos los productos que ha añadido, y realizar acciones sobre ellos antes de finalizar la compra.

Crea el archivo `resources/views/cart/index.blade.php`:

```
touch resources/views/cart/index.blade.php
```

```
@extends('layouts.app')

@section('title', 'Carrito - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
        <div class="mb-8">
            <h1 class="text-3xl font-bold text-gray-900 mb-4">Mi Carrito</h1>
            <p class="text-gray-600">Revisa los productos que has seleccionado.
</p>
        </div>

        @if(!empty($cartItems))
            <div class="bg-white rounded-lg shadow-lg overflow-hidden">
                <div class="overflow-x-auto">
                    <table class="min-w-full divide-y divide-gray-200">
                        <thead class="bg-gray-50">
                            <tr>
                                <th class="px-6 py-3 text-left text-xs font-medium
text-gray-500 uppercase tracking-wider">
                                    Producto
                                </th>
                                <th class="px-6 py-3 text-left text-xs font-medium
text-gray-500 uppercase tracking-wider">
                                    Precio
                                </th>
                                <th class="px-6 py-3 text-left text-xs font-medium
text-gray-500 uppercase tracking-wider">
                                    Cantidad
                                </th>
                                <th class="px-6 py-3 text-left text-xs font-medium
text-gray-500 uppercase tracking-wider">
                                    Total
                                </th>
                            </tr>
                        </thead>
                        <tbody class="bg-white divide-y divide-gray-200">
                            @foreach($cartItems as $item)
                                <tr>
                                    <td class="px-6 py-4 whitespace-nowrap">
                                        <div class="text-sm font-medium text-gray-
900">{{ $item['name'] }}</div>
                                    </td>
                                    <td class="px-6 py-4 whitespace-nowrap text-sm
text-gray-900">
                                        €{{ $item['price'] }}
```

```
                                    </td>
                                    <td class="px-6 py-4 whitespace-nowrap text-sm
text-gray-900">
                                        {{ $item['quantity'] }}
                                    </td>
                                    <td class="px-6 py-4 whitespace-nowrap text-sm
font-medium text-gray-900">
                                        €{{ $item['price'] * $item['quantity'] }}
                                    </td>
                                </tr>
            @endforeach
                        </tbody>
                    </table>
                </div>
        </div>
    @else
        <div class="text-center py-12">
                <p class="text-gray-500 text-lg">Tu carrito está vacío.</p>
                <a href="{{ route('products.index') }}"
                   class="mt-4 inline-block bg-primary-600 text-white px-6 py-3
rounded-lg hover:bg-primary-700 transition">
                    Ver Productos
            </a>
        </div>
    @endif
    </div>
@endsection
```

9. Vista de Contacto

La vista de **contacto** (`resources/views/contact.blade.php`) contendrá un formulario de contacto para que los usuarios puedan enviar mensajes a la tienda.

En esta primera versión, la vista mostrará un mensaje de "En construcción" con un diseño atractivo, animación y un botón para volver al inicio.

Más adelante, aquí se podrá implementar un formulario real con campos como nombre, email, mensaje y validación de datos.

Crea el archivo `resources/views/contact.blade.php`:

```
touch resources/views/contact.blade.php
```

```
@extends('layouts.app')

@section('title', 'Contacto - Mi Tienda')

@section('content')
    <div class="container mx-auto px-6 py-8">
```

```
        <div class="max-w-2xl mx-auto">
            <div class="mb-8 text-center">
                <h1 class="text-3xl font-bold text-gray-900 mb-4">Contacta con
Nosotros</h1>
                <p class="text-gray-600">Estamos aquí para ayudarte. Envíanos un
mensaje.</p>
            </div>

            <div class="bg-white rounded-lg shadow-lg p-8 flex flex-col items-
center justify-center">
                <svg class="w-20 h-20 text-primary-600 mb-4 animate-bounce"
fill="none" stroke="currentColor" stroke-width="2" viewBox="0 0 48 48">
                    <circle cx="24" cy="24" r="22" stroke="currentColor" stroke-
width="4" fill="#e0e7ff"/>
                    <path d="M16 32l8-8 8 8" stroke="currentColor" stroke-
width="3" stroke-linecap="round" stroke-linejoin="round" fill="none"/>
                    <path d="M24 16v8" stroke="currentColor" stroke-width="3"
stroke-linecap="round" stroke-linejoin="round" fill="none"/>
                </svg>
                <h2 class="text-2xl font-bold text-gray-800 mb-2">¡En
Construcción!</h2>
                <p class="text-gray-500 mb-4">Estamos trabajando para traerte esta
funcionalidad muy pronto.</p>
                <a href="{{ route('welcome') }}" class="inline-block mt-2 px-6 py-
2 bg-primary-600 text-white rounded-lg hover:bg-primary-700 transition">
                    Volver al inicio
                </a>
            </div>
        </div>
    </div>
@endsection
```

Estructura Final de Vistas

```
resources/views/
├── layouts/
│   └── app.blade.php          # Layout maestro
├── partials/
│   ├── head.blade.php         # Meta tags y configuración
│   ├── header.blade.php       # Encabezado de página
│   ├── navigation.blade.php   # Navegación principal
│   └── footer.blade.php       # Pie de página
├── components/
│   ├── product-card.blade.php # Componente de tarjeta de producto
│   └── category-card.blade.php # Componente de tarjeta de categoría
├── products/
│   ├── index.blade.php        # Lista de productos
│   └── show.blade.php         # Detalle de producto
├── categories/
│   ├── index.blade.php        # Lista de categorías
│   └── show.blade.php         # Productos de categoría
```

```
├── offers/
│   ├── index.blade.php        # Lista de ofertas
│   └── show.blade.php         # Detalle de oferta con productos
├── cart/
│   └── index.blade.php        # Carrito de compras
├── contact.blade.php          # Página de contacto
└── welcome.blade.php          # Página de bienvenida
```

FASE 9: Probar la Aplicación

1. Verificar las Rutas

Comprobar que cada ruta funciona correctamente y muestra el contenido esperado.

Visita las siguientes URLs en tu navegador:

- **Página de inicio**: `http://localhost`
- **Lista de productos**: `http://localhost/products`
- **Productos en oferta**: `http://localhost/products-on-sale`
- **Producto específico**: `http://localhost/products/1`
- **Lista de categorías**: `http://localhost/categories`
- **Categoría específica**: `http://localhost/categories/1`
- **Lista de ofertas**: `http://localhost/offers`
- **Oferta específica**: `http://localhost/offers/1`
- **Carrito**: `http://localhost/cart`
- **Contacto**: `http://localhost/contact`

> [!CAUTION] Captura de pantalla requerida Realiza una captura de pantalla de la página en cada ruta funcionando. La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible.

2. Navegación

Asegurar que todos los enlaces y botones funcionen correctamente, creando una experiencia de usuario fluida.

Verifica que la navegación funcione correctamente:

- Los enlaces del header te llevan a las páginas correctas
- Los botones "Ver Detalles" funcionan
- Los enlaces de categorías filtran los productos
- Los componentes se muestran correctamente en todas las páginas
- El layout maestro se aplica consistentemente
- La navegación muestra el estado activo correctamente

📋 ENTREGA DE LA PRÁCTICA

Requisitos de Entrega

Para completar esta práctica, debes entregar un **PDF** que incluya las siguientes capturas de pantalla:

> [!NOTE] Navegación Estado activo de navegación funcionando correctamente debe verse en las capturas según la ruta que se esté navegando.

1. Archivo de Rutas Configurado

- Captura de tu terminal mostrando el archivo `routes/web.php` editado con las nuevas rutas
- La captura debe incluir tu prompt `usuario@equipo:~/ruta/proyecto$` visible

2. Datos Mock Personalizados

- Captura de tus archivos de datos mock (`database/data/mock-products.php`, `mock-categories.php`, `mock-offers.php`, etc.) **ajustados a la temática que elegiste** (por ejemplo, libros, ropa, electrónica, etc.)
- La captura debe mostrar claramente cómo personalizaste los nombres, descripciones y campos de los productos, categorías y ofertas según tu temática
- Incluye tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

3. Página de Inicio Funcionando

- Captura de `http://localhost` mostrando la página de bienvenida
- Debe mostrar la temática elegida para tu tienda
- Los productos destacados deben mostrar:
  - Badge naranja en esquina superior derecha con porcentaje de descuento (si tiene oferta)
  - Borde naranja en la tarjeta (si tiene oferta)
  - Precio original tachado y precio final en naranja (si tiene oferta)
  - Nombre de la oferta dinámico (no hardcodeado)
- El botón "🏷️ Ofertas Especiales" debe estar visible
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

4. Página de Detalle de Producto

- Captura de `http://localhost/products/1` (o cualquier ID)
- Debe mostrar la información completa del producto
- Si tiene oferta, debe mostrar:
  - Precio original tachado
  - Precio final con descuento en grande
  - Mensaje "¡Ahorra €X.XX!"
  - Badge con nombre y porcentaje de la oferta
- El botón "Añadir al carrito" debe estar visible
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

5. Página de Lista de Productos

- Captura de `http://localhost/products` mostrando todos los productos
- Debe mostrar la temática elegida para tu tienda
- Los productos deben tener nombres, descripciones y precios personalizados
- Debe mostrar las ofertas de cada producto (badge naranja en esquina)
- Los productos con oferta deben tener borde naranja

- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

6. Página de Productos en Oferta

- Captura de `http://localhost/products-on-sale` mostrando solo productos con oferta
- Debe mostrar el banner rojo/naranja con "🏷️ ¡Productos en Oferta!"
- **CRÍTICO**: Solo deben aparecer productos que tengan `offer_id !== null`
- Debe mostrar el contador: "Mostrando X productos en oferta"
- Todos los productos deben tener destacado visual (badges, bordes, precios tachados)
- NO debe haber productos sin oferta en esta vista
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

7. Página de Lista de Categorías

- Captura de `http://localhost/categories` mostrando todas las categorías
- Debe mostrar las categorías personalizadas según tu temática
- Cada categoría debe tener nombre y descripción personalizados
- Los enlaces "Ver Productos" deben funcionar correctamente
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

8. Página de Productos por Categoría

- Captura de `http://localhost/categories/1` (o cualquier ID)
- Debe mostrar solo los productos de esa categoría
- Navegación entre categorías funcionando
- Los productos con ofertas deben mostrar destacado visual completo:
  - Badge de descuento dinámico en esquina
  - Precio tachado + precio final
  - Nombre de oferta desde los mocks (no hardcodeado)
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

9. Página de Lista de Ofertas

- Captura de `http://localhost/offers` mostrando todas las ofertas
- Debe mostrar las ofertas personalizadas según tu temática
- Cada oferta debe tener nombre, descripción y porcentaje de descuento
- Los enlaces "Ver Productos" deben funcionar correctamente
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

10. Página de Detalle de Oferta

- Captura de `http://localhost/offers/1` (o cualquier ID)
- Debe mostrar el header destacado con el porcentaje de descuento
- Debe mostrar todos los productos que tienen aplicada esa oferta
- Los productos deben mostrarse usando el componente ProductCard con destacado visual
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

11. Página del Carrito

- Captura de `http://localhost/cart` mostrando el carrito de compras
- Debe mostrar los productos añadidos al carrito
- Debe mostrar precios, cantidades y totales
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

12. Página de Contacto

- Captura de `http://localhost/contact` mostrando la página de contacto
- Debe mostrar el mensaje "En construcción" con diseño atractivo
- El botón "Volver al inicio" debe funcionar correctamente
- La captura debe incluir tu terminal con el prompt `usuario@equipo:~/ruta/proyecto$` visible

Instrucciones de Entrega

1. **Formato**: Entregar un único archivo PDF con todas las capturas
2. **Organización**: Incluir un título para cada captura explicando qué muestra
3. **Calidad**: Las capturas deben ser claras y legibles
4. **Personalización**: La tienda debe reflejar claramente la temática elegida con datos personalizados
5. **Funcionalidad**: Todas las rutas y navegación deben funcionar correctamente
6. **Terminal**: Todas las capturas deben incluir el prompt del terminal visible
7. **Completitud**: Deben mostrarse todas las vistas creadas (inicio, productos, categorías, ofertas, carrito, contacto)
8. **Datos personalizados**: Los productos, categorías y ofertas deben estar adaptados a tu temática específica

Plataforma de Entrega

Entrega en Aules