How to Build a Shopping Cart in Laravel

A shopping cart is one of the most common features in e-commerce applications. In Laravel, you can implement it in a clean, modular way using controllers, models, Blade views, and session or database storage. In this guide, we'll build a minimal yet extensible shopping cart system that supports adding products, updating quantities, removing items, persisting to database (optional), and rendering a clean UI. We'll also cover validation, middleware guards, and testing to ensure reliability.

Database Setup for Products

Products should be stored in the database. For simplicity, our cart will use the `products` table.

```
php artisan make:model Product -mCode language: Bash (bash)
```

```
// database/migrations/xxxx_xx_xx_create_products_table.php
Schema::create('products', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->decimal('price', 10, 2);
    $table->unsignedInteger('stock')->default(0);
    $table->timestamps();
});Code language: PHP (php)
```

Seed your products table with some demo items for testing.

Session-Based Cart Structure

We'll store cart items inside the session as an array keyed by product ID:

```
[
    1 => ['name' => 'Laptop', 'price' => 999.99, 'quantity' => 2],
    3 => ['name' => 'Mouse', 'price' => 25.00, 'quantity' => 1],
]Code language: PHP (php)
```

This structure keeps cart operations simple. Later, you can swap to a database-backed cart for authenticated users without changing much logic.

Cart Controller & Routes

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

Route::prefix('cart')->name('cart.')->group(function () {
    Route::get('/', [CartController::class, 'index'])->name('index');
    Route::post('/add/{product}', [CartController::class,
'add'])->name('add');
    Route::post('/update/{product}', [CartController::class,
'update'])->name('update');
    Route::post('/remove/{product}', [CartController::class,
'remove'])->name('remove');
    Route::post('/clear', [CartController::class,
'clear'])->name('clear');
});Code language: PHP (php)
```

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

use App\Models\Product;
```

```
use Illuminate\Http\Request;

class CartController extends Controller
{
    public function index(Request $request)
    {
        $cart = $request->session()->get('cart', []);
        $total = collect($cart)->sum(fn ($item) => $item['price'] *
$item['quantity']);
        return view('cart.index', compact('cart','total'));
    }

    public function add(Request $request, Product $product)
    {
        $cart = $request->session()->get('cart', []);

        $qty = (int) $request->input('quantity', 1);
        if ($qty < 1) $qty = 1;

        if (isset($cart[$product->id])) {
            $cart[$product->id]['quantity'] += $qty;
        } else {
            $cart[$product->id] = [
                'name' => $product->name,
                'price' => $product->price,
                'quantity' => $qty,
            ];
        }

        $request->session()->put('cart', $cart);
        return back()->with('success', 'Product added to cart.');
    }

    public function update(Request $request, Product $product)
    {
        $cart = $request->session()->get('cart', []);
        if (isset($cart[$product->id])) {
            $cart[$product->id]['quantity'] = max(1, (int)
```

```
$request->input('quantity', 1));
            $request->session()->put('cart', $cart);
        }
        return back()->with('success', 'Cart updated.');
    }

    public function remove(Request $request, Product $product)
    {
        $cart = $request->session()->get('cart', []);
        unset($cart[$product->id]);
        $request->session()->put('cart', $cart);
        return back()->with('success', 'Item removed.');
    }

    public function clear(Request $request)
    {
        $request->session()->forget('cart');
        return back()->with('success', 'Cart cleared.');
    }
}Code language: PHP (php)
```

The controller uses session storage to track cart items and provides endpoints for all core operations.

Blade UI for Cart

```
<!-- resources/views/cart/index.blade.php -->
<h2>Shopping Cart</h2>

@if(empty($cart))
    <p>Your cart is empty.</p>
@else
    <table>
```

```
        <thead>
<tr><th>Product</th><th>Qty</th><th>Price</th><th>Total</th><th></th><
/tr>
        </thead>
        <tbody>
        @foreach($cart as $id => $item)
            <tr>
                <td>{{ $item['name'] }}</td>
                <td>
                    <form method="POST" action="{{
route('cart.update',$id) }}">
                        @csrf
                        <input type="number" name="quantity" value="{{
$item['quantity'] }}" min="1" />
                        <button>Update</button>
                    </form>
                </td>
                <td>${{ number_format($item['price'],2) }}</td>
                <td>${{ number_format($item['price'] *
$item['quantity'],2) }}</td>
                <td>
                    <form method="POST" action="{{
route('cart.remove',$id) }}">
                        @csrf
                        <button>Remove</button>
                    </form>
                </td>
            </tr>
        @endforeach
        </tbody>
    </table>

    <p><strong>Total:</strong> ${{ number_format($total,2) }}</p>

    <form method="POST" action="{{ route('cart.clear') }}"> @csrf
        <button>Clear Cart</button>
    </form>
@endifCode language: PHP (php)
```

The Blade template loops through session items, calculates totals, and offers controls for updating and removing products.

Adding to Cart from Product List

```
<!-- resources/views/products/index.blade.php -->
@foreach($products as $product)
    <div>
        <h3>{{ $product->name }}</h3>
        <p>${{ number_format($product->price,2) }}</p>

        <form method="POST" action="{{ route('cart.add',$product) }}">
            @csrf
            <input type="number" name="quantity" value="1" min="1" />
            <button>Add to Cart</button>
        </form>
    </div>
@endforeachCode language: PHP (php)
```

This simple UI lets users add products to the cart with quantity input. The form posts to `cart.add`, which handles insertion into the session.

Persisting Cart for Authenticated Users

For logged-in users, you may want to store carts in the database. Create a `carts` table linked to users and move session data into persistent storage on login or checkout. This allows users to resume shopping across devices.

```
Schema::create('carts', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')->constrained()->onDelete('cascade');
    $table->json('items');
    $table->timestamps();
});Code language: PHP (php)
```

Sync session data to the user's cart record on login or checkout for persistence.

Testing the Cart

```
// tests/Feature/CartTest.php
use App\Models\Product;
use Tests\TestCase;

class CartTest extends TestCase
{
    public function test_can_add_product_to_cart()
    {
        $product = Product::factory()->create();

        $this->post('/cart/add/'.$product->id, ['quantity' => 2])
            ->assertSessionHas('success');

        $this->assertEquals(2,
session('cart')[$product->id]['quantity']);
    }

    public function test_can_update_and_remove_cart_item()
    {
        $product = Product::factory()->create();

        $this->post('/cart/add/'.$product->id);
```

```
        $this->post('/cart/update/'.$product->id, ['quantity' => 5]);

        $this->assertEquals(5,
session('cart')[$product->id]['quantity']);

        $this->post('/cart/remove/'.$product->id);
        $this->assertArrayNotHasKey($product->id, session('cart',
[]));
    }
}Code language: PHP (php)
```

Tests confirm that cart operations work correctly. Extend tests to include edge cases like invalid quantities and stock limits.

Wrapping Up

We built a minimal shopping cart in Laravel using session storage, complete with add, update, remove, and clear functionality. The Blade UI renders cart contents and totals, and we discussed persisting carts for authenticated users. This foundation can be extended with product stock checks, coupons, checkout workflows, and payment integrations.

What's Next

Continue building your Laravel e-commerce toolkit:

- Building a Simple SaaS Billing System with Laravel and Cashier
- How to Integrate Stripe Payments in Laravel

- Handling File Uploads and Image Storage in Laravel