SSP 2 - Security Documentation and Implementation

Name: Thisalma Alwis
CB No: CB016928

Table Of content

Git Hub Link:- https://github.com/thisalma/Delica_Website.git

1. SQL Injections Prevention

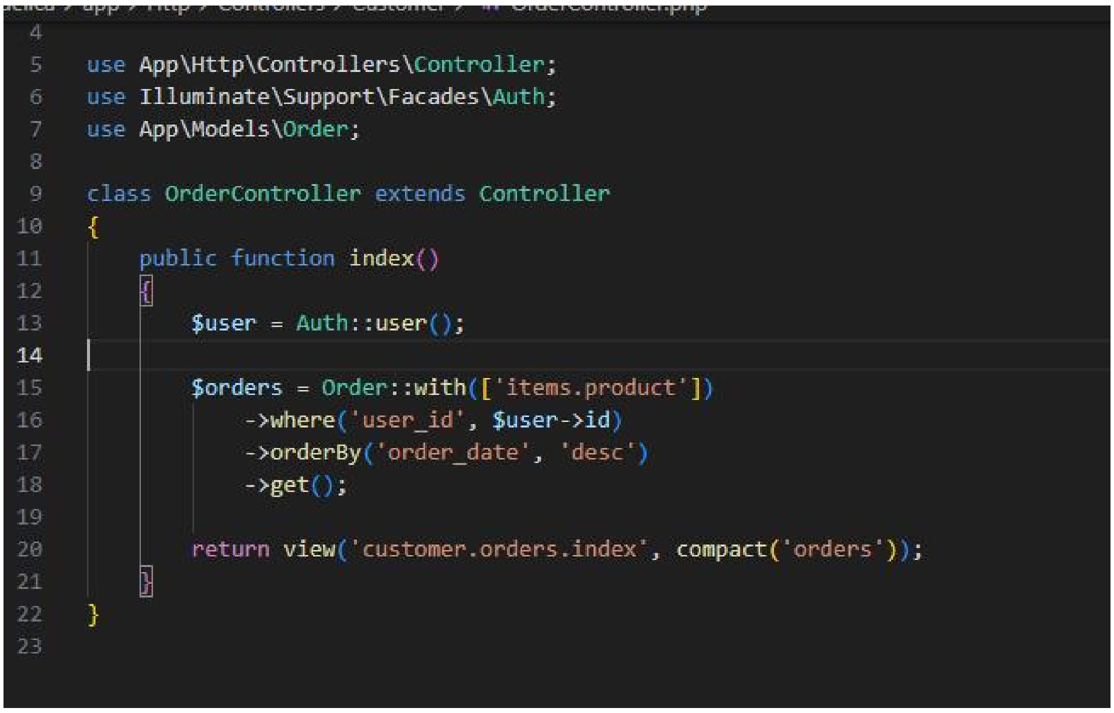
SQL Injection – attackers try to manipulate queries by injecting malicious input

Mitigation in Laravel:

- Using Eloquent ORM instead of raw SQL queries.
- Route Model Binding automatically validates IDs in URLs.
- No string concatenation inside queries.
- Optional: $request->validate() for sanitizing user input before saving.

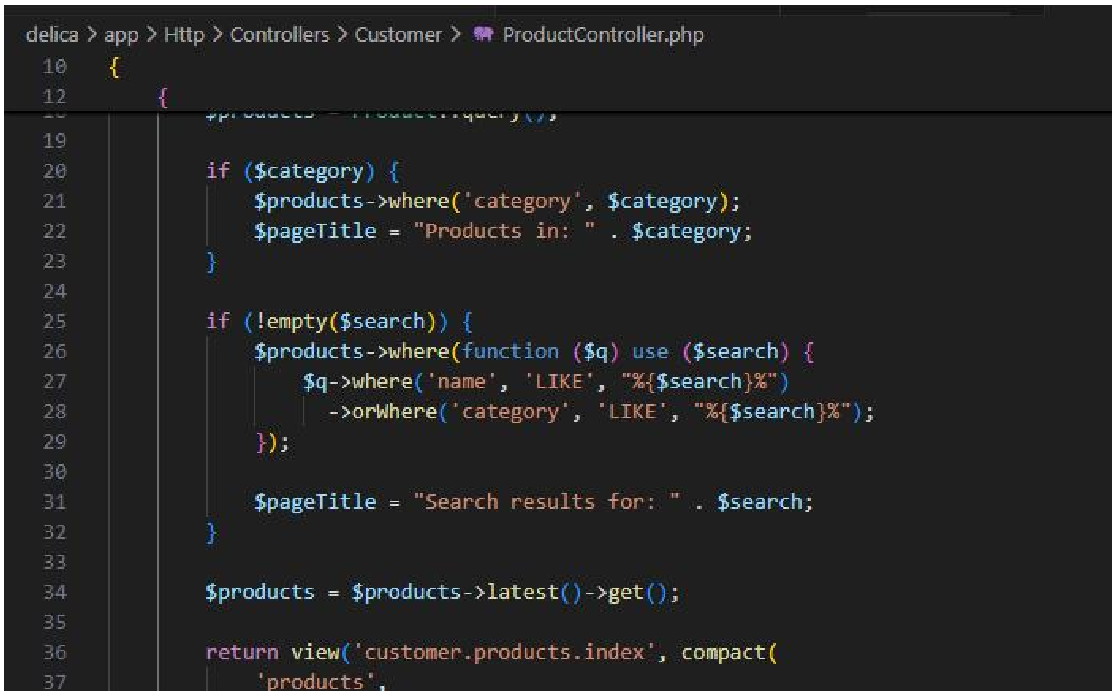
Screenshots:

Fetching orders in OrderController:



```php

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Auth;
use App\Models\Order;

class OrderController extends Controller
{
    public function index()
    {
        $user = Auth::user();

        $orders = Order::with(['items.product'])
            ->where('user_id', $user->id)
            ->orderBy('order_date', 'desc')
            ->get();

        return view('customer.orders.index', compact('orders'));
    }
}

```

Searching products in ProductController:


delica > app > Http > Controllers > Customer > ProductController.php

```
    {
        {
            $products = Product::query();

            if ($category) {
                $products->where('category', $category);
                $pageTitle = "Products in: " . $category;
            }

            if (!empty($search)) {
                $products->where(function ($q) use ($search) {
                    $q->where('name', 'LIKE', "%{$search}%")
                      ->orWhere('category', 'LIKE', "%{$search}%");
                });

                $pageTitle = "Search results for: " . $search;
            }

            $products = $products->latest()->get();

            return view('customer.products.index', compact(
                'products',
```

Test Case Table:

Test Case	Input	Expected Result	Actual Result	Pass/Fail
Fetch Orders (OrderController)	Try injecting ' OR 1=1 -- in user ID	Only logged-in user’s orders are fetched; injection ignored	Only logged-in user’s orders fetched	Pass
Search Products (ProductController)	Search input: ' OR 1=1 --	No SQL injection occurs; products matching the string are searched safely	Products filtered safely; injection ignored	Pass
Search Products (ProductController)	Search input: Electronics' OR '1'='1	No SQL injection occurs; search results filtered safely	Search results safe; only matching products returned	Pass

2. Password Hashing

Exploit Name: Plain-text passwords stored in database.

Mitigation in Laravel:

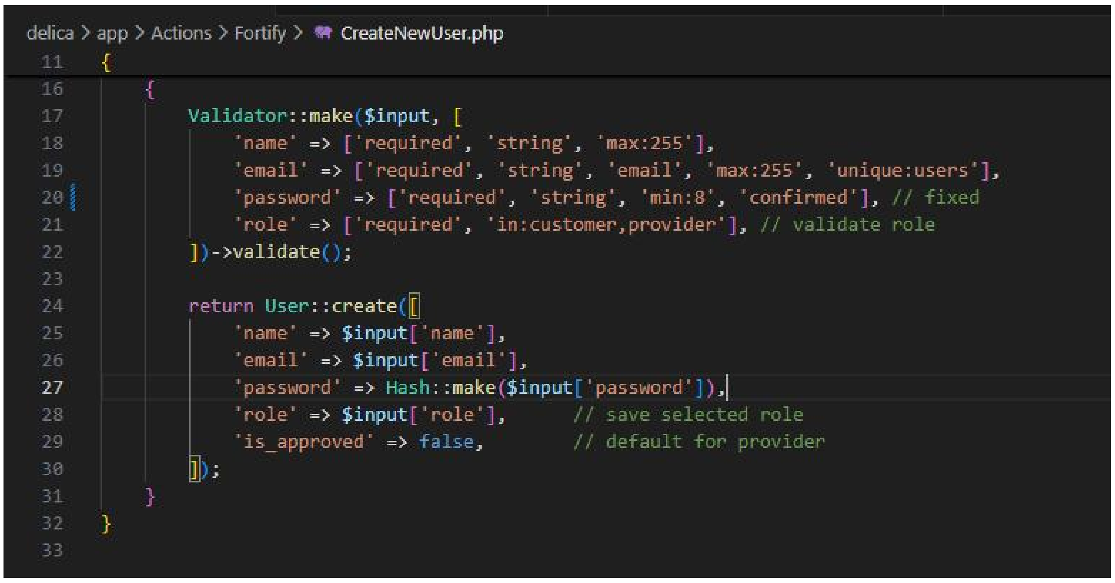
Laravel uses Hash::make(), which applies bcrypt hashing by default.
Passwords are never stored in plain text in the database.
Laravel automatically verifies hashed passwords during login using the Auth system.

Evidence for Implemenation:

- CreateNewUser.php
- Database screen shot

Screenshots

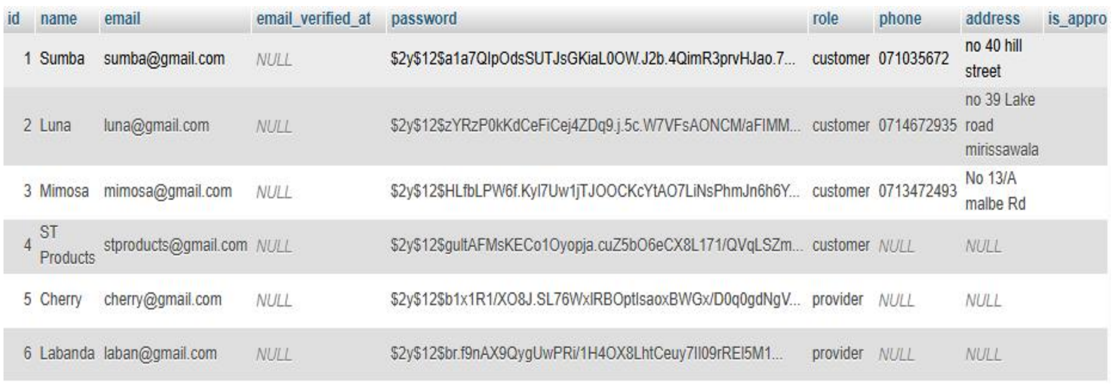
Password hashing implementation inside CreateNewUser.php:



delica > app > Actions > Fortify > CreateNewUser.php

```
{
    {
        Validator::make($input, [
            'name' => ['required', 'string', 'max:255'],
            'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],
            'password' => ['required', 'string', 'min:8', 'confirmed'], // fixed
            'role' => ['required', 'in:customer,provider'], // validate role
        ])->validate();

        return User::create([
            'name' => $input['name'],
            'email' => $input['email'],
            'password' => Hash::make($input['password']),
            'role' => $input['role'],      // save selected role
            'is_approved' => false,        // default for provider
        ]);
    }
}
```

Database screenshot showing hashed password



id	name	email	email_verified_at	password	role	phone	address	is_appro
1	Sumba	sumba@gmail.com	NULL	$2y$12$a1a7QlpOdsSUTJsGKiaL0OW.J2b.4QimR3prvHJao.7...	customer	071035672	no 40 hill street	
2	Luna	luna@gmail.com	NULL	$2y$12$zYRzP0kKdCeFiCej4ZDq9.j.5c.W7VFsAONCM/aFIMM...	customer	0714672935	no 39 Lake road mirissawala	
3	Mimosa	mimosa@gmail.com	NULL	$2y$12$HLfbLPW6f.Kyl7Uw1jTJOOCKcYtAO7LiNsPhmJn6h6Y...	customer	0713472493	No 13/A malbe Rd	
4	ST Products	stproducts@gmail.com	NULL	$2y$12$gultAFMsKECo1Oyopja.cuZ5bO6eCX8L171/QVqLSZm...	customer	NULL	NULL	
5	Cherry	cherry@gmail.com	NULL	$2y$12$b1x1R1/XO8J.SL76WxlRBOptlsaoxBWGx/D0q0gdNgV...	provider	NULL	NULL	
6	Labanda	laban@gmail.com	NULL	$2y$12$br.f9nAX9QygUwPRi/1H4OX8LhtCeuy7Il09rREI5M1...	provider	NULL	NULL	

Test Case Table

Test Case	Input	Expected Result	Actual Result	Pass/Fail
User Registration – Password Hashing	Password: mypassword123	Password should be stored in hashed (bcrypt) format, not plain text	Password stored as $2y$10$... hashed value in database	Pass
User Registration – Plain Text Check	Password: test@1234	Plain text password should not appear in database	Plain text not visible; only hashed value stored	Pass
Login Authentication	Email + correct password	User should authenticate successfully using hashed password comparison	User logged in successfully	Pass
Login Authentication – Wrong Password	Email + incorrect password	Authentication should fail	Login rejected	Pass

3. Route Protection (Middleware)

Exploit: Unauthorized access to protected routes (e.g., /admin/dashboard, /provider/dashboard).

Mitigation in Laravel:

- Auth middleware (auth) ensures only logged-in users can access protected routes.
- Role-based middleware (role:customer, role:provider) restricts routes based on user roles.
- Admin guard (auth:admin) secures admin routes separately from normal users.
- Email verification (verified) ensures only verified users can access certain routes.

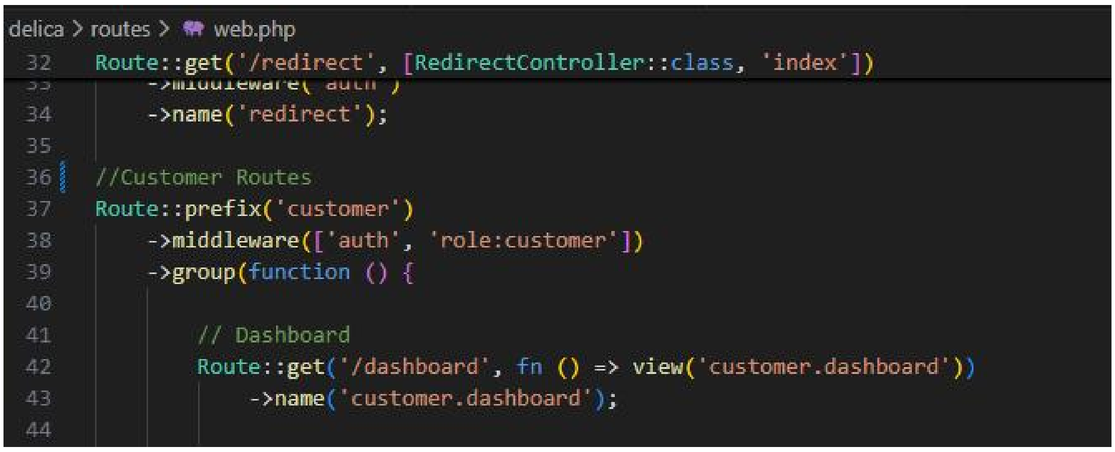
Evidence of Implementation

- Customers cannot access provider or admin routes.
- Providers cannot access customer or admin routes.
- Admin routes are protected using auth:admin.
- Non-authenticated users are redirected to login automatically.

ScreenShot

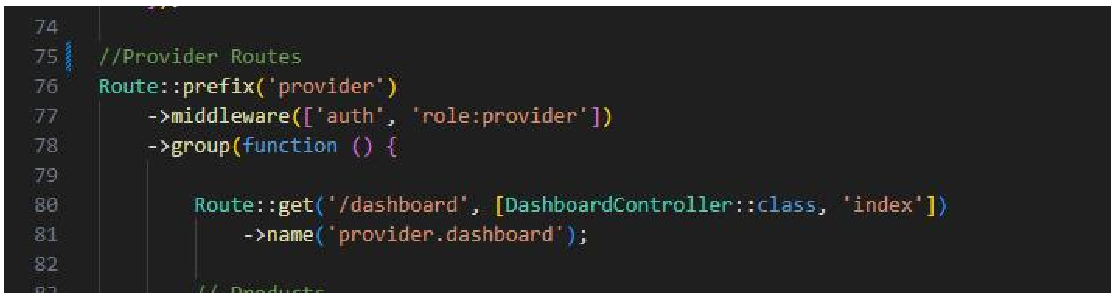
Middleware applied in web.php for customer, provider, and admin routes.

customer



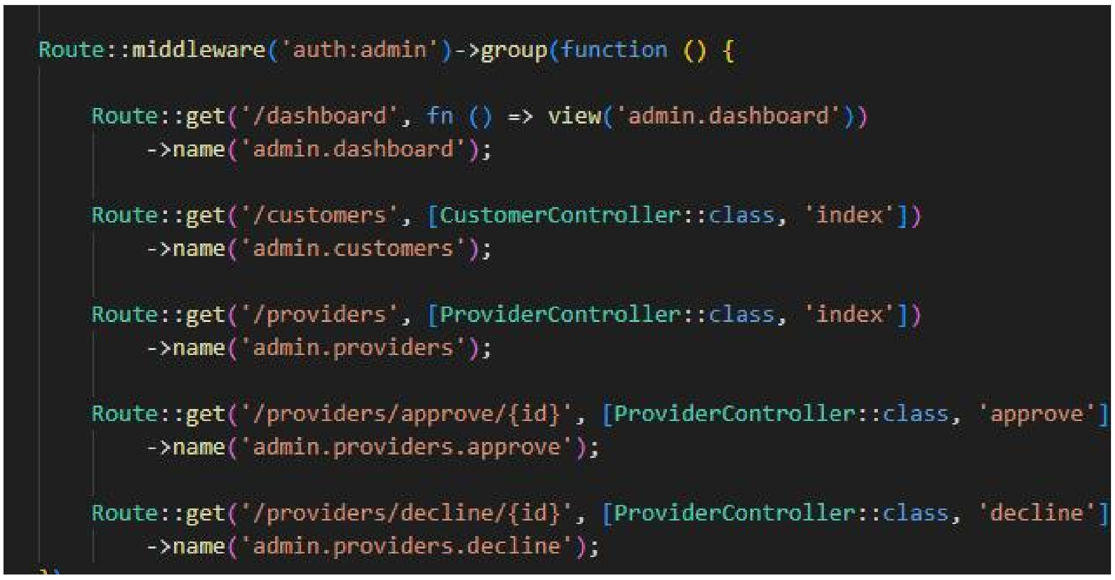
```
delica > routes > web.php
Route::get('/redirect', [RedirectController::class, 'index'])
    ->middleware('auth')
    ->name('redirect');

//Customer Routes
Route::prefix('customer')
    ->middleware(['auth', 'role:customer'])
    ->group(function () {

        // Dashboard
        Route::get('/dashboard', fn () => view('customer.dashboard'))
            ->name('customer.dashboard');

```

Provider



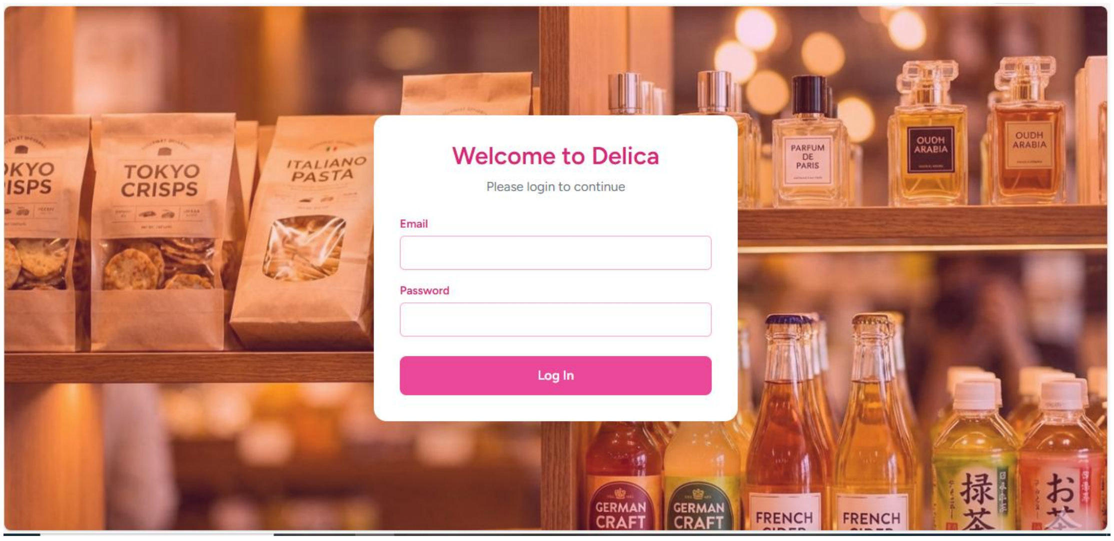
```

//Provider Routes
Route::prefix('provider')
    ->middleware(['auth', 'role:provider'])
    ->group(function () {

        Route::get('/dashboard', [DashboardController::class, 'index'])
            ->name('provider.dashboard');

        // Products
```

Admin



```
Route::middleware('auth:admin')->group(function () {

    Route::get('/dashboard', fn () => view('admin.dashboard'))
        ->name('admin.dashboard');

    Route::get('/customers', [CustomerController::class, 'index'])
        ->name('admin.customers');

    Route::get('/providers', [ProviderController::class, 'index'])
        ->name('admin.providers');

    Route::get('/providers/approve/{id}', [ProviderController::class, 'approve']
        ->name('admin.providers.approve');

    Route::get('/providers/decline/{id}', [ProviderController::class, 'decline']
        ->name('admin.providers.decline');
```

Redirect to login page when accessing protected routes while not authenticated



Test case Table

Test Case	Input	Expected Result	Actual Result	Pass/Fail
Access Customer Dashboard (Authenticated Customer)	Logged in as customer → /customer/dashboard	Page loads successfully	Page loaded	Pass
Access Customer Dashboard (Non-Authenticated User)	Not logged in → /customer/dashboard	Redirected to login page	Redirected to login	Pass
Access Provider Dashboard (Authenticated Provider)	Logged in as provider → /provider/dashboard	Page loads successfully	Page loaded	Pass
Access Provider Dashboard (Customer Trying to Access)	Logged in as customer → /provider/dashboard	Access denied / redirect	Redirected / access denied	Pass
Access Admin Dashboard (Authenticated Admin)	Logged in as admin → /admin/dashboard	Page loads successfully	Page loaded	Pass
Access Admin Dashboard (Customer or Provider Trying to Access)	Logged in as customer/provider → /admin/dashboard	Access denied / redirect	Redirected / access denied	Pass
Default Dashboard (Unauthenticated)	Not logged in → /dashboard	Redirected to login	Redirected to login	Pass
Default Dashboard (Authenticated & Verified User)	Logged in & verified → /dashboard	Redirected to /redirect	Redirected	Pass

4. CSRF Protection

Exploit:
Cross-Site Request Forgery (CSRF) – a malicious website that can submit forms on behalf of an authenticated user without their consent (e.g., changing profile information like name, phone, or address).

Mitigation in Laravel:

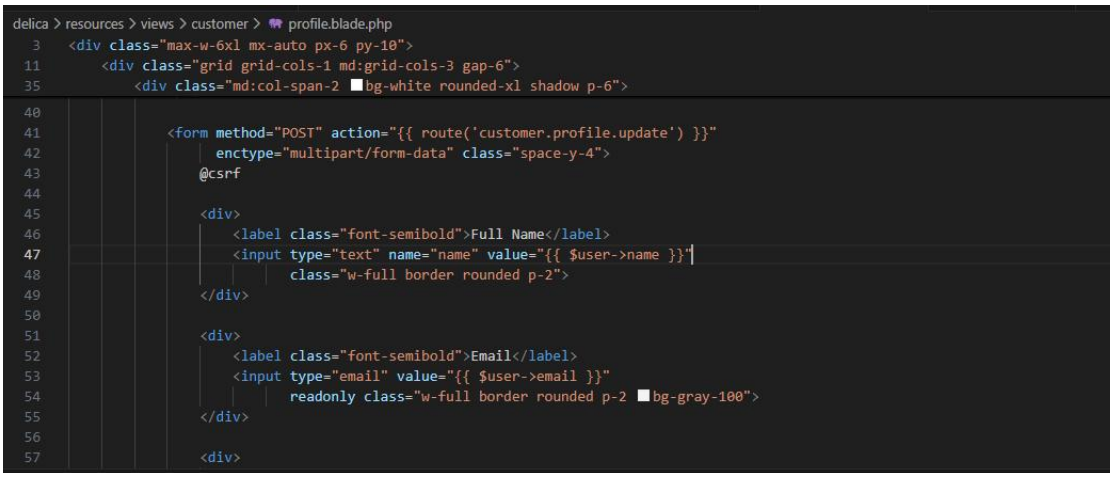
- Laravel automatically protects all state-changing requests using CSRF tokens.
- The @csrf Blade directive adds a hidden token field to the form.
- Requests submitted without a valid CSRF token are rejected by Laravel with a 419 Page Expired error.
- Token validation is handled automatically by Laravel's VerifyCsrfToken middleware.

Evidence for Implementation

- The @csrf directive ensures that a unique token is included with every profile update request.
- Laravel automatically validates the token before processing the request.
- Attempting to submit the form without @csrf results in a 419 Page Expired error, demonstrating that CSRF protection is active.

Screenshots

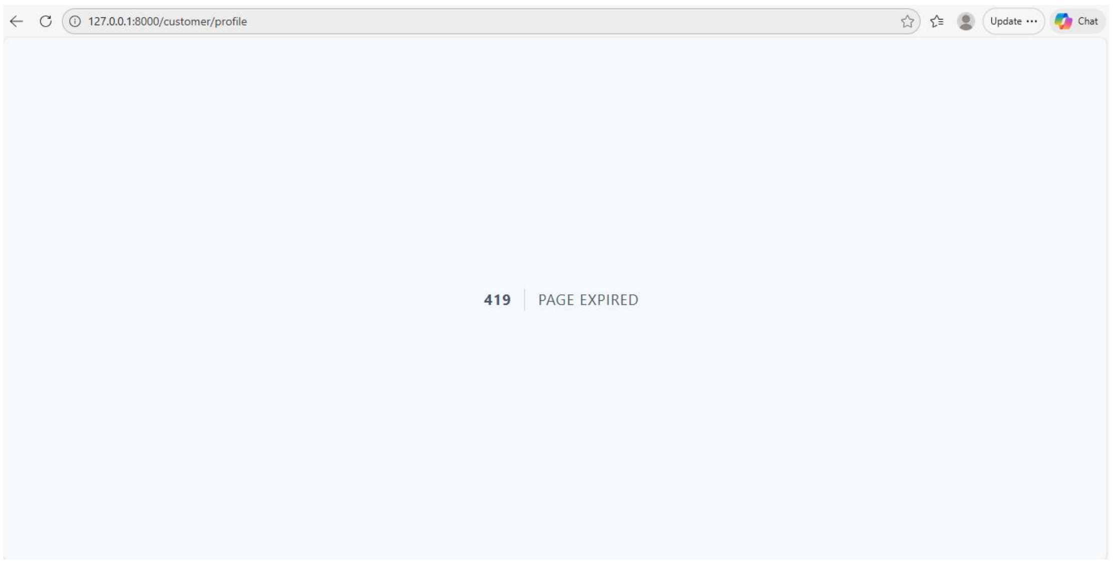
Profile update Blade file showing @csrf directive.



delica > resources > views > customer > profile.blade.php

```
<div class="max-w-6xl mx-auto px-6 py-10">
    <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
        <div class="md:col-span-2 bg-white rounded-xl shadow p-6">

            <form method="POST" action="{{ route('customer.profile.update') }}"
                enctype="multipart/form-data" class="space-y-4">
                @csrf

                <div>
                    <label class="font-semibold">Full Name</label>
                    <input type="text" name="name" value="{{ $user->name }}"
                           class="w-full border rounded p-2">
                </div>

                <div>
                    <label class="font-semibold">Email</label>
                    <input type="email" value="{{ $user->email }}"
                           readonly class="w-full border rounded p-2 bg-gray-100">
                </div>

                <div>
```

419 error page when the form is submitted without a CSRF token



419 | PAGE EXPIRED

Test case Table:

Test Case	Input	Expected Result	Actual Result	Pass/Fail
Submit profile form with CSRF token	Update Name, Phone, Address, and Profile Picture, then submit	Profile updated successfully	Profile updated successfully	Pass
Submit profile form without CSRF token	Remove @csrf from Blade, fill form, and submit	Request blocked, 419 Page Expired error	419 Page Expired shown	Pass
Submit profile form with invalid CSRF token	Manually change the token value in the hidden input and submit	Request blocked, 419 Page Expired error	419 Page Expired shown	Pass
Access profile page (GET request)	Open /customer/profile in browser	Form page loads normally	Form page loaded	Pass

5. Additional Security Sanctum API

Sanctum API Security

- All cart-related routes (/api/cart-json, /api/cart-json/add/{product}, /api/cart-json/remove/{product}) are protected using Laravel Sanctum.
- Only authenticated users with a valid Bearer token can access these routes.
- Each user can generate a personal access token to interact with the API (e.g., testing via Postman).
- Tokens can be revoked if compromised, ensuring secure API access.
- This prevents unauthorized users from manipulating the cart or performing actions on behalf of another user.

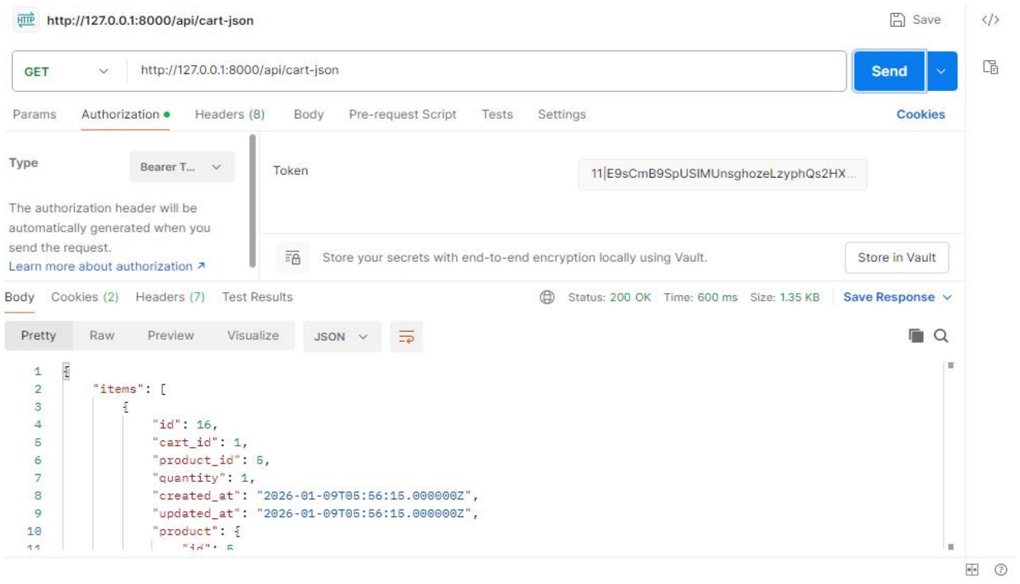
Role-Based Access Control (RBAC)

- The system distinguishes users by roles: Admin, Owner, and Customer.
- Cart operations are only accessible to users with the Customer role.
- Laravel Gates or Policies can be applied to ensure that customers cannot access admin routes or modify other users' carts, preventing privilege escalation.
- This enforces separation of privileges and protects sensitive operations like product management or user management.

Implementation Notes

- Website routes (/customer/cart) are handled by Livewire for authenticated users, ensuring session-based security.
- API routes (/api/cart-json) provide JSON responses for external clients and testing, secured via Sanctum token authentication.
- The combination of Sanctum tokens and RBAC ensures that both web and API requests are authenticated and authorized correctly.

Screenshots



GET http://127.0.0.1:8000/api/cart-json

Type: Bearer T...

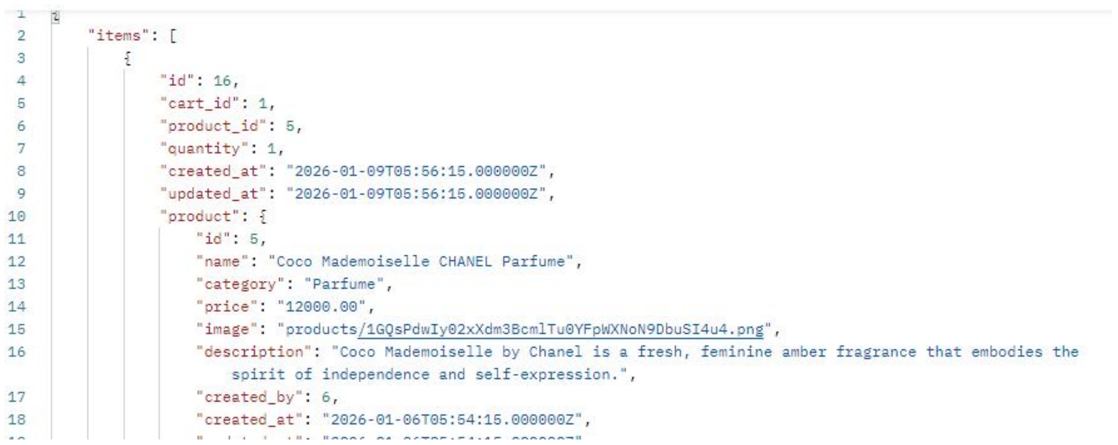
The authorization header will be automatically generated when you send the request.
Learn more about authorization ↗

Token: 11|E9sCmB9SpUSIMUnsghozeLzyphQs2HX...

Store your secrets with end-to-end encryption locally using Vault.

Status: 200 OK Time: 600 ms Size: 1.35 KB

```
{
    "items": [
        {
            "id": 16,
            "cart_id": 1,
            "product_id": 5,
            "quantity": 1,
            "created_at": "2026-01-09T05:56:15.000000Z",
            "updated_at": "2026-01-09T05:56:15.000000Z",
            "product": {
```



```
{
    "items": [
        {
            "id": 16,
            "cart_id": 1,
            "product_id": 5,
            "quantity": 1,
            "created_at": "2026-01-09T05:56:15.000000Z",
            "updated_at": "2026-01-09T05:56:15.000000Z",
            "product": {
                "id": 5,
                "name": "Coco Mademoiselle CHANEL Parfume",
                "category": "Parfume",
                "price": "12000.00",
                "image": "products/1GQsPdwIy02xXdm3BcmlTu0YFpWXNoN9DbuSI4u4.png",
                "description": "Coco Mademoiselle by Chanel is a fresh, feminine amber fragrance that embodies the
                    spirit of independence and self-expression.",
                "created_by": 6,
                "created_at": "2026-01-06T05:54:15.000000Z",
```

Test Case Table

Test Case	Input	Expected Result	Actual Result	Pass/Fail
1. Access cart without token	GET /api/cart-json (No Authorization header)	Access denied, 401 Unauthorized	401 Unauthorized	Pass
2. Access cart with valid Customer token	GET /api/cart-json with Bearer token of a Customer	Returns JSON with cart items and total	Returns JSON with correct cart data	Pass
3. Add product to cart without token	POST /api/cart-json/add/1 (No Authorization header)	Access denied, 401 Unauthorized	401 Unauthorized	Pass
4. Add product to cart with valid Customer token	POST /api/cart-json/add/1 (Bearer token)	Product added, returns JSON with message and cart_item	Product added, JSON returned correctly	Pass

Test Case	Input	Expected Result	Actual Result	Pass/Fail
5. Remove product from cart without token	POST /api/cart-json/remove/1 (No Authorization header)	Access denied, 401 Unauthorized	401 Unauthorized	Pass
6. Remove product from cart with valid Customer token	POST /api/cart-json/remove/1 (Bearer token)	Product removed, JSON returns message: Product removed from cart.	Product removed, JSON returned correctly	Pass
7. Add product with invalid ID	POST /api/cart-json/add/999 (Bearer token)	Returns 404 Not Found or error message	404 Not Found	Pass
8. Access cart with Admin token	GET /api/cart-json with Admin token	Access denied (Customer-only operation) or empty cart	Access denied or empty cart	Pass
9. Remove product not in cart	POST /api/cart-json/remove/999 (Bearer token)	Returns message Product removed from cart. or appropriate 404	Message returned, no error	Pass
10. Check total calculation	GET /api/cart-json after adding multiple products	Total in JSON matches sum of product prices × quantities	Total matches	Pass