DIY Electro

Complete User Documentation

Routes → Controllers → Services → Views

Full narrative reference for every storefront module: browsing, cart, coupons, DIY points, checkout, orders, client area, and content pages.

Generated for internal reference

DIY Electro User Documentation

DYI-CLONE — COMPLETE FRONTEND DOCUMENTATION

Full step-by-step narrative of all frontend views, features, and modules for the DYI e-commerce storefront. Covers how each page works (route → controller → service → view) and what it contains. No code snippets — only narrative logic flow.

DETAILED EXPLANATION (PINNED AT TOP)

1. What this storefront is. DYI is a full-featured e-commerce storefront built on Laravel (Blade) with a completely custom frontend layer. Every public page is rendered by a controller in `app\Http\Controllers\Web\*`, which delegates all business logic to a service class in `app\Services\*`, and finally passes data to a Blade template in `resources\views\frontend\*`. The frontend does NOT use Laravel's default authentication guard for customers — all client identity lives in the session (`session('user_id')`, `session('user_name')`) plus a persistent cookie (`persistent_user_id`) that gives ~1 year "remember me" behaviour.

2. The global data pipeline. Almost every frontend controller injects `SettingService` and calls `getSettingSeoScripts()`, which returns one merged array containing `setting`, `seo`, `body`, `head`, and `categories`. This is why every Blade view automatically has `$setting` (site config), `$seo`, `$head` / `$body` (analytics/script tags), and `$categories` (the full 4-level category tree used for the mega-menu) available without any per-page work. Site-wide changes made in the admin appear on all frontend pages on the very next request.

3. The page pipeline (route → controller → service → view). For each module the documentation below follows the same 4-stage walkthrough. (a) **Route** — the URL pattern registered in `routes/web.php` (each route has a name like `home.product-detail`, `cart.show`, `order.store`). (b) **Controller** — the thin HTTP layer that reads the request, checks login/session state, calls the service, and returns either a view or JSON. (c) **Service** — the brain: all pricing, discount, coupon, wallet, order, and wishlist math lives in service classes (`CartService`, `CheckoutService`, `ProductService`, `CouponService`, `DiyPointsService`, `AuthService`, etc.). (d) **View** — the Blade template under `resources/views/frontend` that renders the page using the data prepared by the service.

4. How auth and guest gating work. Clients log in via the header login modal or `/user-register`. After login the session stores `user_id` and `user_name`, and a `persistent_user_id` cookie is queued for 525600 minutes (~1 year). Guest-only features (cart, wishlist, checkout, dashboard, tracking) check `session('user_id')`; if missing, the controller either redirects with `->with('show_login', true)` (the header auto-opens the login modal) or returns an HTTP 401 JSON payload with `login_required=true` for AJAX calls so the frontend JS can open the modal itself. The

admin panel is the only place that uses Laravel's real auth guard (`Auth::routes(['register'=>false])`).

5. The pricing & discount engine. Every money figure flows through the same rules: base price = offer price if set and lower, else original price; a Dealer-approved client gets a dealer % applied only when it is lower than the normal price; GST is split out by each product's `product_gst` rate; extra discount (from the single-row `ExtraDiscount` config) triggers above a cart-value threshold; coupons (admin coupons or user-generated PINs) subtract a flat amount; DIY points redeem wallet balance; free shipping zeroes shipping when the total clears the `FreeShipping` threshold. The exact same engine runs in the cart, checkout, order placement, success page, failed page, and invoice so numbers never drift.

6. Order lifecycle. Placing an order (`CheckoutService::placeOrder`) recomputes everything from the DB, branches by payment method (Razorpay / Wallet / COD / Pickup), creates the `Order` + `OrderList` rows, decrements stock, sends emails via BrevoMailService, and — only for fully-paid orders — records coupon usage, awards referral rewards and earned DIY points, and credits wallets. Payment cancellation creates a recorded Cancelled order and redirects to the Failed page so the user can retry while their cart is preserved.

7. How to read this document. Start with Part 1 (shared infrastructure) because every other module depends on it. Then read Parts 2–6 in order. Each STEP lists the exact routes and files, a numbered controller-logic-flow narrative, what the page actually contains, and a detailed Features list (at least 8 lines per module) explaining what the module does, why it exists, and how to change its behaviour.

TABLE OF CONTENTS

PART 6 — CONTENT & STATIC PAGES - Step 1: About / Company Overview - Step 2: Terms / Privacy / FAQ - Step 3: Blogs - Step 4: DIY Shorts - Step 5: Services - Step 6: Career - Step 7: Contact & Product Enquiry / Notify-Me / Order-back - Step 8: Newsletter

PART 1 — FOUNDATION & SHARED INFRASTRUCTURE

The infrastructure shared by every public page: the global data merge, the header/footer layouts, the session-based client auth, and guest gating.

STEP 1 Global Data Pipeline (SettingService)

File: `app/Services/SettingService.php` **Used in:** Every frontend controller render

Controller Logic Flow:

1. **Injection** → Every Web controller constructor injects `SettingService` through Laravel's container (e.g. `ProductController`, `HomeController`, `CartController`, `CheckoutController`, `PageController`).
2. **Service** → `SettingService::getSettingSeoScripts()` (`app/Services/SettingService.php`) returns the packaged array `['setting', 'seo', 'body', 'head', 'categories']` which controllers merge into every view via `array_merge($data, $common)`.
3. **Memoized getters** → Each getter lazy-loads once and caches in a protected property:
   - `getSetting()` → `Setting::latest()->first()` (site-wide config: logos, favicon, scroll text, phone/email, social links, store address for pickup, checkout notice)
   - `getSeo()` → `Seo::latest()->first()`
   - `getBodyScripts()` / `getHeadScripts()` → all `BodyScript` / `HeadScript` rows
   - `getCategories()` → full category tree `ProductCategory::with('subcategories.child_categories.sub_child_categories')`
4. **Per-request caching** → Because the service is resolved once per request, repeated calls hit the in-memory cache (no duplicate DB queries).
5. **Downstream effect** → `$setting` drives the entire storefront chrome (header/footer content, mega-menu, pickup store info, checkout notice) and `$categories` drives the header mega-menu and footer popular searches.

This is the reason every Blade view automatically has `$setting`, `$seo`, `$head`, `$body`, `$categories` available.

Plain-English Walkthrough: When a visitor opens any page, the browser calls a route like `/` or `/products`. The controller asks `SettingService` for the common data, which loads the site settings, SEO row, analytics scripts, and the full category tree from the database as one package and merges it into the page data before returning the view. The header then uses `$categories` to draw the mega-

menu and `$setting` to draw the logo/contact info — the view never runs its own queries. If the admin edits a phone number or adds a tracking script, the very next page load shows it automatically.

How to use: Never touch this file per-page. Change site-wide values (contact info, SEO, analytics head/body scripts, category structure) in the admin — they appear on all frontend pages next request.

Features: - **Single source of truth for site config** — One `Setting` row (and one `Seo` row) powers logos, favicon, contact phone/email, social links, scrolling announcement text, the store pickup address, and the checkout notice banner, so a single admin edit updates every page simultaneously with zero code changes. - **Automatic script injection** — Every admin-managed head script (analytics, meta, pixel) and body script is rendered on every page via the `$head` / `$body` loops in the layout, so marketing pixels and SEO tags stay consistent across the whole storefront without editing templates. - **One-call category tree** — The full 4-level category hierarchy (`categories → subcategories → child_categories → sub_child_categories`) is eager-loaded once and reused by the header mega-menu, mobile drawer, footer popular searches, and filter sidebars, guaranteeing the navigation is always in sync everywhere. - **Zero per-page boilerplate** — Controllers just call `array_merge($data, $common)` ; views never re-query settings, SEO, or categories, which removes duplicated code and keeps page controllers tiny and focused on page-specific work. - **Memoization prevents N+1 queries** — Because each getter caches its result in a property on the resolved service instance, dozens of pages calling the same getters within one request cause exactly one query per getter, keeping the storefront fast under traffic.

STEP 2 Header (layouts/header.blade.php)

File: `resources/views/frontend/layouts/header.blade.php` **Used in:** Every public page (included via `@include('frontend.layouts.header')`)

Controller Logic Flow:

1. Reads the global `$setting` , `$head` , `$body` , `$categories` , `$populars` supplied by the SettingService merge; also consumes `$isGuest` , `$isLoggedIn` , `$cartCount` , `$wishCount` computed by the including controller.
2. The `<head>` block writes the favicon/preload logo from `$setting` , loops `$head` to inject analytics/meta/script tags, and renders a loading spinner.
3. Desktop navbar: logo, search form (posts to `/search`), account dropdown, cart & wishlist icons with live badges, and a 4-level category mega-menu (`$categories` → `subcategories` → `child_categories` → `sub_child_categories`).
4. A Services dropdown loops `$services` linking to `/service/{slug}` . Session blocks: `@if(session('user_name'))` shows the client dropdown (Dashboard, Orders, Wishlist, Cart, Logout); guests see a login trigger.
5. Mobile sidebar mirrors the category accordion (same 4-level tree), side links, a newsletter form posting to `newsletter.subscribe` , social icons, and a bottom mobile nav (Home / Wishlist / Cart / Account). Wishlist/cart links call `showLoginAlert(event)` when `$isGuest` .

6. Global AJAX add-to-cart handler lives at the bottom: POSTs to `cart.add`, updates `.cart-count-badge` from JSON `cart_count`, shows a toast; on `login_required` it opens the login modal. Flash `session('success')` / `session('error')` / `$errors` blocks render styled alerts.
7. A login modal (`#loginModal`) is included and auto-opens when `session('show_login')` or `?user-login` is present (the guest-gating redirect hook).

What the page contains: 1. Top announcement/scrolling-text bar from `$setting` 2. Desktop navbar with mega-menu (4 category levels) + Services dropdown 3. Account dropdown (login state aware) 4. Cart + Wishlist icons with live AJAX-synced count badges 5. Mobile drawer sidebar with accordion categories + newsletter form 6. Bottom mobile nav bar (Home / Wishlist / Cart / Account) 7. Login modal (auto-open on `show_login`)

Plain-English Walkthrough: When any page loads, the header partial renders first. It draws the logo, the search box, and the mega-menu from the category tree. If a user is logged in, the account icon expands to a dropdown of account links; otherwise it shows a login trigger. When the user clicks “Add to Cart” on any product card anywhere on the site, the shared header script POSTs to `/add-to-cart`, gets back the new cart count, updates the badge, and shows a small toast. If the server replies “login required”, the script automatically opens the login modal so the user can sign in without leaving the page.

How to use: Automatic. To force login for a guest, redirect with `->with('show_login', true)` or append `?user-login`. Badges stay in sync because add/remove endpoints return counts.

Features: - **One global AJAX add-to-cart handler** — Every product card’s “Add to Cart” button wires into the header’s shared JS, which POSTs to `cart.add`, reads the JSON `cart_count`, updates the header badge, and shows a toast. A `login_required` response opens the login modal, so the flow works identically across home, shop, search, offer, and category pages with no duplicated JS. - **Deep 4-level mega-menu** — The desktop navbar builds its dropdown from the full category tree (`categories → subcategories → child_categories → sub_child_categories`), letting shoppers drill from a top-level category down to a specific sub-sub-sub category in one hover without leaving the header. - **Login-state-aware account area** — Logged-in users see a dropdown with Dashboard, Orders, Wishlist, Cart, and Logout; guests see a login trigger that opens the modal. The same block powers the mobile drawer and bottom nav, so the UX is consistent on every device. - **Global search entry point** — The header search form posts to `/search`, so a query typed anywhere on the site always lands on the Search module which applies the full filter + relevance logic (see Part 2, Step 4). - **Guest-gating hook baked into the chrome** — When a controller redirects with `show_login`, the header auto-opens `#loginModal`; when AJAX returns `login_required`, the shared JS opens the same modal. This single mechanism protects cart, wishlist, checkout, and dashboard from guests without per-page work. - **Live badge sync from server truth** — Cart and wishlist counts come from the server (`cart_count` in add-to-cart responses, `GET /counts`), not local storage, so badges always match the database even after login/logout or across devices.

STEP 3 Footer (layouts/footer.blade.php)

File: `resources/views/frontend/layouts/footer.blade.php` **Used in:** Every public page (included via `@include('frontend.layouts.footer')`)

Controller Logic Flow:

1. Reads the same global `$setting`, `$isGuest`, `$isLoggedIn`, `$cartCount`, `$wishCount`.
2. Renders the footer logo via `ImageHelper::getPhotoUrl($setting->footer_logo)`, a newsletter card (email input posting to `newsletter.subscribe`), quick links, and a contact block from `$setting->phone_1` / `$setting->email_1`.
3. Social icons render only when the matching link exists in `$setting`: facebook, arattai, instagram, youtube, twitter, linkedin. App-store/play-store link uses `$setting->playstore_link`.
4. “Popular searches” section loops `$populars` producing direct `/search` links.
5. A mobile bottom-nav (desktop-hidden) shows Home, Wishlist, Cart, Account — with the same `$isGuest` gating via `showLoginAlert()` and badges from `$wishCount` / `$cartCount`.

What the page contains: 1. Footer logo + tagline 2. Newsletter subscribe card 3. Quick links + popular searches 4. Contact block (phone / email / address) 5. Conditional social icons + app/play-store links 6. Mobile bottom-nav with guest gating 7. Copyright strip

Plain-English Walkthrough: At the bottom of every page the footer draws the logo, contact info, popular searches, and social icons straight from the settings. The newsletter box lets a visitor type an email and click Subscribe — the form POSTs to `/subscribe` and the page refreshes showing a success message, or a validation error if the email is already subscribed. On mobile, a bottom bar with Home / Wishlist / Cart / Account appears; tapping Wishlist or Cart while logged out triggers the login prompt.

How to use: Fully automatic. Configure footer logo, contact numbers, emails, social links, play store link, and popular searches in admin settings.

Features: - **Admin-driven footer content** — Logo, tagline, phone numbers, emails, address, and the app/play-store link all come from the single `$setting` record, so marketing can restyle the footer without touching Blade templates. - **Conditional social icons** — Each social icon (Facebook, Arattai, Instagram, YouTube, Twitter, LinkedIn) renders only when its link is filled in the settings, which avoids ugly empty placeholders when a channel is not yet launched. - **Popular searches as one-click search links** — The `$populars` list renders as direct links into `/search`, helping shoppers jump straight to commonly-sought terms and acting as a lightweight SEO/discovery feature. - **Built-in newsletter capture** — The footer (and the mobile drawer) embeds the newsletter form that posts to `newsletter.subscribe`, feeding the subscriber list the admin uses for campaigns (see Part 6, Step 8). - **Mobile bottom nav with guest gating** — Home, Wishlist, Cart, and Account appear on a desktop-hidden bottom bar; guests hitting Wishlist/Cart get the `showLoginAlert` login prompt, and the badges reflect the same server counts as the desktop header. - **Consistent flash handling** — Success/error session flashes and `$errors` render in styled alert blocks, so validation failures from the newsletter form (or any redirected form) display cleanly inside the footer layout.

STEP 4 Auth — Login / Register / Google / Forgot Password

Files: `AuthController` (`app/Http/Controllers/Web/AuthController.php`), `AuthService` (`app/Services/AuthService.php`), `GoogleAuthController`, views `frontend/auth/login.blade.php`, `frontend/auth/login-modal.blade.php`, `frontend/client/forgot-password.blade.php`, `frontend/client/reset-password.blade.php`

Controller Logic Flow:

1. **Routes** (`routes/web.php`):
   - `/user-login` (line 536) → `AuthController@login` → redirects to `/?user-login` (opens the modal)
   - `/user-register` (line 537) → `register` → renders `frontend.auth.login` with a sliding sign-in/sign-up panel
   - POST `/login/store` (line 539) → `loginStore` (register)
   - POST `/login/check` (line 540) → `loginCheck` → `loginForm` (login)
   - `/client/logout` (line 541) → `logout`
   - `/client/switch-back` (line 542) → `switchBack`
   - Google (lines 560-565), bridge (line 535), forgot/reset (lines 644-653)
2. **Register flow** (`loginStore`): sanitizes name/email/phone/title, validates (name letters-only, email `rfc,dns` + unique, password min 8, phone regex `^(0\d{10}|[6789]\d{9})$` + unique, title in Regular/Dealer), then calls `AuthService::register()` inside a DB transaction.
3. **`AuthService::register`**: reads `Referal::first()` for `signup_points`, resolves `referred_by` by matching `Client.referral_code` against the `reff` query param or `referred_by` input, creates the Client (password hashed + also stored plain), `firstOrCreate`s a Wallet, credits signup points via a `WalletTransaction` (`payment_method='Signupbonus'`), and emails a welcome via `ClientRegisteredMail`.
4. **Post-register**: session regenerates, `user_id` / `user_name` are stored, the intended URL is honored unless it points to admin/login, then redirect (or JSON if AJAX).
5. **Login flow** (`loginForm`): sanitizes email, validates, `AuthService::login()` finds by email and `Hash::check`s the password; differentiates “email does not exist” vs “password mismatch”; on success regenerates session, stores `user_id` / `user_name`, and queues the `persistent_user_id` cookie (525600 minutes ≈ 1 year).
6. **Logout**: forgets `user_id` / `user_name`, queues cookie forget for `persistent_user_id`, redirects home with a success flash.
7. **Google**: `/auth/google` → redirects to Google; `/auth/google/callback` → Socialite stateless — logs in an existing Client (by `google_id` or email) or creates one with a random password,

`Signupbonus` wallet credit, and a welcome email. `/auth/google-signup/callback` → `googleSignUp` explicitly rejects existing-email signups. Both set the persistent cookie.

8. **Forgot/Reset**: `/forgot-password` GET/POST → stores a 64-char token in `password_resets` and emails the reset link. `/reset-password/{token}` GET shows the form; POST validates email + confirmed password + token via `validateResetToken`, then `resetPassword` updates the password (bcrypt + plain copy) and deletes the token.

What the page contains: 1. Sliding sign-in / sign-up panel (`frontend.auth.login`) 2. Login modal (header, `login-modal.blade.php`) for guest gating 3. Google Sign-in / Google Sign-up buttons 4. Forgot password form + reset password form (client area layouts) 5. Debounced live phone/email uniqueness checks (post to `checkPhoneUnique` / `checkEmailUnique`)

Plain-English Walkthrough: A new visitor clicks the Account icon and chooses Register (or visits `/user-register`). The form asks for name, email, phone, password, and lets them paste a referral code. As they type, the page live-checks that the email and phone aren't taken. On submit the controller validates everything, the service creates the Client, creates a wallet, credits the signup bonus, and emails a welcome — then the user is logged in immediately and sent back where they were. A returning visitor clicks Login, types email + password; the service checks the credentials, starts a session, and drops a one-year cookie. If they forget the password, they request a reset link, open it from email, and set a new password.

How to use: Users open the Account icon → modal or `/user-register`. Referral links append `?reff=CODE`. After login the persistent cookie keeps users logged in ~1 year. Forgot-password flows are email-based.

Features: - **Dual login surface (modal + full page)** — The same sign-in/sign-up form powers both the quick header modal (for guest gating) and the dedicated `/user-register` page, so users can authenticate in-context or on a focused page; both share identical validation and service logic. - **One-tap Google sign-in and sign-up** — Two separate OAuth callbacks let the storefront log in an existing account (`/auth/google/callback`) or explicitly create a new one (`/auth/google-signup/callback`, which rejects existing emails), preventing accidental account merges while keeping onboarding friction low. - **Automatic referral tagging** — Registering through a `?reff=CODE` link (or the hidden referred_by field) resolves the referrer's `referral_code`, stores the relationship on the new Client, and credits the referrer's wallet via the signup/referral reward rules (see Part 4, Step 2), making the referral loop fully automatic. - **Welcome bonus on first signup** — `AuthService::register` reads the signup points value from the `Referal` settings and writes a `Signupbonus` wallet transaction, so every new account starts with a funded wallet usable at checkout. - **Distinct, friendly error messages** — The login service separates "email does not exist" from "password mismatch", so users understand exactly what to fix instead of seeing a generic credential error. - **Robust password recovery** — Forgot/reset uses a 64-char single-use token emailed to the user, validates the token on submit, updates both the bcrypt hash and the plain-text copy the legacy flow reads, and deletes the token to prevent reuse. - **Persistent session cookie** — Successful login, registration, and Google auth all queue the `persistent_user_id` cookie (1-year lifetime), giving returning visitors a seamless "still logged in" experience across browser restarts.

STEP 5 Session, Persistent Cookie & Guest Gating

How it works:

1. The storefront does NOT use Laravel's default auth guard for clients. `Auth::routes(['register'=>false])` (routes/web.php:48) exists only for the admin panel. All client identity is `session('user_id')` / `session('user_name')`.
2. Guest gating pattern: controllers read `session('user_id')`; if absent they redirect with `->with('show_login', true)->with('error', ...)` or return HTTP 401 JSON with `login_required=true` for AJAX. The header/footer pick up `show_login` and auto-open `#loginModal`.
3. `HomeController@index` and `DashboardController@index` validate the session user still exists; if the account was deleted, they clear the session and queue `Cookie::forget('persistent_user_id')`.
4. The persistent cookie is set on login/register/Google and re-set on return visits (home recomputes counts); it provides "remember me" style auto-login.
5. `switchBack` is for admin impersonation: if `session('admin_id')` exists, it clears client keys and returns to the admin clients list.
6. `bridgeLogin` accepts a Sanctum token from another app, finds/creates a matching Client, and establishes the session (cross-app login).

Plain-English Walkthrough: The storefront tracks who is logged in through the session, not through Laravel's normal login system. When a guest tries to open the cart, the controller finds no user id in the session and redirects with a "show login" flag, so the header pops the login modal. On AJAX actions (like adding to cart or wishlist) the same check returns a `login_required` error and the frontend JS opens the modal instead. A one-year cookie keeps the user remembered. If the admin deletes an account, the homepage detects the leftover session, wipes it, and forgets the cookie.

How to use: Whenever you build a page that needs a user, check `session('user_id')`. For cart/wishlist/checkout/dashboard/track pages, gate guests with `show_login`. Clear both session keys and the cookie on any account-destruction path.

Features: - **Separate client identity from admin auth** — The storefront never touches Laravel's web guard for customers, so client sessions and the admin panel are fully isolated; deleting or changing an admin account cannot affect shoppers, and vice-versa. - **Uniform guest-gating hook** — One convention (`show_login` flash for page loads, `login_required` JSON for AJAX) is understood by the shared header JS, which opens the login modal automatically — so every protected page uses identical UX instead of bespoke "please login" screens. - **Self-healing stale sessions** — Home and Dashboard verify the session user still exists in the DB; when the account has been deleted they clean the session keys and forget the persistent cookie, preventing ghost sessions and orphaned carts/wishlists. - **Remember-me via persistent cookie** — The `persistent_user_id` cookie (1 year) re-establishes the session on return visits and is re-queued on login/register/Google and on home visits, giving a seamless cross-visit experience. - **Admin impersonation escape hatch** — `switchBack` clears the impersonated client keys when `admin_id` is present and returns the admin to the clients list,

so support staff can safely step out of a customer's session. - **Cross-app bridge login** — `bridgeLogin` accepts a Sanctum token, matches or creates the Client, and starts a session — enabling SSO-style login from companion apps without exposing credentials.

PART 2 — PUBLIC BROWSING & DISCOVERY

What visitors and logged-in users see when browsing the catalog.

STEP 1 Home Page (home/index.blade.php)

Route: `GET /` → `HomeController@index` (routes/web.php:488) → `HomeService::getHomeData()` + `SettingService::getSettingSeoScripts()`

Controller Logic Flow:

1. **Controller** → `HomeController@index` reads `session('user_id')`, verifies the Client still exists (else cleans session + cookie), calls `getHomeData($userId)`, merges the common SEO/scripts data, returns `view('frontend.home.index', ...)`.
2. **Service** → `HomeService::getHomeData()` (`app/Services/HomeService.php`):
    1. **Single-row content**: `About::latest()->first()`, `AboutSectionTwo`, `HeadingChanger`, `ShippingCardDynamic::all()`.
    2. **Ordered collections**: `Banner` by `sort_order` asc (carousel), `ContentPage::latest()` (blog), `Testimonial::latest()`, `OfferPromotion::latest()` (offer banners), `Popular::all()`, `Brand::all()`, `TopProductsWeekend::all()`.
    3. **Product buckets by status flag**: Active first 12 (`$products`), ComingSoon 4, CustomerFav 4, TrendingPro 4, ComboPack 4, BestPro 4, Offer 4, Active latest 4 (`$latests`).
    4. **Category grouping**: `$categoryProducts` = last 24 Active products eager-loaded with category, grouped by `category.name` (powers the “shop by category” carousel while capping DOM size). `$groupedSubMainCategories` collects distinct `sub_main_category` strings with their related `ProductCategory` records.
    5. **Login enrichment**: cart count, wishlist count, `$wishlistedProductIds`; for `Dealer Approved` clients sets `$clientStatus` + `$clientDiscount`; `$notifiedProductIds` from `Enquiry` rows matching the client email (Notify-Me buttons render “requested”).
    6. Returns ~35 compacted variables (banners, categories, product buckets, testimonials, brands, blog, shipping cards, dealer/wishlist/notified state).

Plain-English Walkthrough: A visitor lands on the homepage. The controller checks whether they are logged in (so it can personalize), then the service gathers hero banners, eight product buckets, categories, testimonials, brands, and blog posts in one call. The page slides through the hero banners, shows the shipping-strip icons, then carousels of products. Logged-in users see filled wishlist hearts, correct cart counts, and — for Dealer-approved users — dealer prices on the cards. Any section that has no content simply hides itself.

What the page contains: 1. Hero banner carousel (Owl Carousel) with click-through links 2. Shipping-card dynamic strip (icons + titles) 3. Category carousel (“shop by category”) 4. Customer Favorites carousel (desktop Owl / mobile Swiper, with wishlist + quick-add) 5. Our Products carousel (main Active bucket) 6. Latest Products carousel 7. Additional product buckets (Trending, Combo Packs, Best, Offer) with section titles from `HeadingChanger` 8. Testimonials, brands, blog cards, popular searches, scroll-text marquee 9. Empty sections self-hide

Features: - **One endpoint assembles the whole landing page** — `getHomeData` returns roughly 35 prepared variables (banners, eight product buckets, category groupings, testimonials, brands, blog cards, shipping cards, dealer/wishlist/notified state) so the view is pure rendering with no queries, keeping the heaviest page on the site fast and predictable. - **Status-flag-driven product buckets** — Products are surfaced into the “Our Products”, “Latest”, “Customer Favorites”, “Trending”, “Combo Packs”, “Best”, and “Offer” carousels purely by their `status` value, so merchandising is done in the admin without code changes. - **Category-scoped “shop by category” carousel** — The last 24 active products are grouped by their category name, letting shoppers browse a category directly from the homepage while the DOM size stays capped for performance. - **Personalization for logged-in users** — Wishlist hearts are pre-filled, cart count is correct, Notify-Me buttons show as “requested” when an enquiry already exists, and Dealer-approved clients see their discount applied on cards immediately. - **Dual carousel engine** — Desktop carousels run Owl Carousel while mobile switches to Swiper, giving touch-friendly swipe behaviour on phones and hover/arrow navigation on desktops. - **Self-hiding empty sections** — Banners, testimonials, brands, blog, and product buckets that have no content simply do not render, so the page never shows broken or empty blocks even when the catalog is sparse.

STEP 2 Shop / Product Listing (products/index.blade.php)

Route: `GET /products/{category_name?}/{sub_category_name?}/{child_category_name?}/{sub_child_category_name?}` → `ProductController@index` (routes/web.php:490-493, name `home.products`)

Controller Logic Flow:

1. **Controller** → `ProductController@index` calls `ProductService::getFilteredProducts($request, $category_name, $sub_category_name, $child_category_name, $sub_child_category_name)`.
2. If `$request->ajax()`: renders each product to HTML via `view('frontend.products.card', ...)` and returns `{products_html, last_page}` for AJAX filtering/pagination. Otherwise returns the full view.
3. **Service** → `ProductService::getFilteredProducts()` (`app/Services/ProductService.php`):
    1. Loads `Size::all()`, `Brand::all()`, full category tree, `Popular::all()`.

2. Resolves category params: URL segments override the query params; slugs matched by scanning the table in PHP and filtering by resolved ids (`category_id` , `sub_category_id` , `pro_child_category_id` , `pro_sub_child_category_id`).
3. Reads filter inputs: `sizes[]` , `brands[]` , `minPrice` , `maxPrice` , `sortBy` .
4. Base query only includes sellable statuses: `['Active','CustomerFav','TrendingPro','ComboPack','BestPro','Offer']` .
5. Size filter uses `whereHas('details', size_id in ...)` OR `whereHas('sizeOnlyDetails', size_id in ...)` . Brand filter is `whereIn('brand', ...)` .
6. Price filter: offer-priced products must be `BETWEEN` on `offer_price` , others on `price` (DECIMAL-cast).
7. Sort: `high_to_low` / `low_to_high` order by `COALESCE(NULLIF(offer_price,0),price)` ; `newest` by `created_at desc` ; default `latest()` .
8. **Dealer/wishlist enrichment**: for `Dealer Approved` users computes `$clientDiscount` and `$wishlistedProductIds` . Paginates 9/page; each product mapped with `final_price_per_unit` (offer if >0 else original), and dealer price applied only when lower (`is_dealer_price_applied` , `dealer_price_calculated`).

Plain-English Walkthrough: A visitor opens `/products` or clicks a category in the mega-menu. The page shows a product grid with a filter sidebar. When they select a size, tick a brand, or drag the price slider, an AJAX request is sent; the server re-runs the product query with those filters and returns fresh card HTML, which replaces the grid — no page reload. The filters are kept in the address bar, so the filtered view can be copied and shared. Dealer users automatically see their discount on every card.

What the page contains: 1. Breadcrumb (Home / Products / sub-main / category) 2. Sidebar filters: category accordion (4 levels), “Shop by size” checkboxes, “Shop by Brand” checkboxes, price range slider + inputs, Filter / Clear buttons, results counter 3. Product grid (rendered from `products.card` partials) 4. AJAX loader container + “Load more” / infinite-scroll pagination 5. Mobile: native sort select + slide-in filter drawer mirroring the sidebar 6. Empty state (“No products found”)

Features: - **Zero-page-reload filtering** — The `JS fetchFilteredProducts` routine POSTs filter/sort state, gets back rendered product HTML + the last page number, and swaps the grid in place, so shoppers can filter by size, brand, and price without losing scroll position or reloading the page. - **Deep-category URL navigation** — The route accepts up to four slug segments (`category/sub_category/child_category/sub_child_category`), each resolved to its ID by scanning the table, so both the mega-menu links and typed URLs drive the exact same filtered listing. - **Sellable-only product base** — The base query restricts results to `Active, CustomerFav, TrendingPro, ComboPack, BestPro, Offer` statuses, so InActive/coming-soon items never leak into the shop grid. - **Offer-aware price filtering** — Products with an offer price are filtered on `offer_price` while normal products use `price` , so the price slider reflects what shoppers will actually pay, and sorting uses `COALESCE(NULLIF(offer_price,0),price)` to rank by the real selling price. - **Dealer-aware pricing on every card** — Dealer-approved users get their discount computed (`dealer_price_calculated`) and shown only when it beats the store price (`is_dealer_price_applied`), keeping wholesale pricing consistent across the whole listing. - **Mobile-first filter UX** — On small screens the sidebar collapses into a slide-in drawer and a native sort select appears above the grid, so the full desktop filtering power

is preserved on phones without squeezing the layout. - **Shareable, persistent filter URLs** — Filters live in query params, so a shopper can copy a filtered URL and send it to someone else who lands on the same filtered view.

STEP 3 Category Pages (categories/show.blade.php & categories/products.blade.php)

Routes: `GET /product-category/{name}` → `CategoryController@category` (routes/web.php:517, view `categories/show`); `GET /category/{main_category?}` → `ProductController@categoryProducts` (routes/web.php:530, view `categories/products`); `GET /products/category/{main_category}` (routes/web.php:485)

Controller Logic Flow:

1. **`CategoryService::getBySlug($name)`** (`app/Services/CategoryService.php`): loads sizes/brands/categories/populars, finds `ProductCategory::where('name',$name)->firstOrFail()`, queries Active products in that category, paginates 12 → `categories/show`.
2. **`CategoryService::categoryProducts($request, $sub_main_category, $main_category)`** is the AJAX-driven flow behind `categories/products`: resolves the main category by name → id, validates the requested `subcategory_id` belongs to it (else null), and builds the query with `when()` clauses: main category filter, price range (offer-or-price between), subcategory filter (specific id or `whereIn` all subcategories when none selected), sizes (`whereHas`), brands (`whereIn`), and sort (COALESCE offer/price).
3. Pagination: 9/page for the AJAX flow, 12 for `getBySlug`, 20 for legacy `showCategory`.

Plain-English Walkthrough: A visitor clicks a main category link. The service resolves the category to its id and applies the same filters as the shop (sizes, brands, price, sort). Choosing a subcategory first verifies it belongs to the selected main category, then re-queries and re-renders the grid via AJAX while updating the URL so the view stays shareable. A simpler slug-based category page also exists for plain links that just list that category's products.

What the page contains: 1. Category header + breadcrumb 2. Filter sidebar (category accordion, sizes, brands, price) — same as shop 3. In-page search box 4. Product grid with AJAX re-render on filter change 5. `activeCategoryUrl` rewrite so the URL reflects the chosen subcategory

Features: - **Dedicated category browsing** — Both a simple slug-based category page (`/product-category/{name}`) and a fully interactive AJAX category page (`/category/{main_category}`) exist, so internal links and marketing URLs each get an appropriate experience. - **Validated subcategory scoping** — When a subcategory filter is requested the service verifies it actually belongs to the selected main category before using it; otherwise it falls back to `whereIn` all subcategories, preventing empty or mismatched results from bad URL parameters. - **Same filter engine as the shop** — Price, size, brand, and sort filters reuse the exact `when()` query builders from `ProductService`, so behaviour (offer-aware price ranges, sellable statuses, COALESCE sorting) is identical across category pages and the main shop. - **AJAX re-render with URL sync** — Choosing a subcategory rewrites

`activeCategoryUrl` so the address bar reflects the current selection while the grid updates instantly, keeping results shareable without a full reload. - **In-page search** — A search box is included on the category page so shoppers can refine within the category without leaving it, complementing the sidebar filters. - **Consistent pagination sizes** — 9 items per page for the AJAX flow keeps infinite-scroll light, while the simpler non-AJAX fallback pages use 12 and 20 per page for legacy compatibility.

STEP 4 Search Page (search/index.blade.php)

Route: `GET /search?query=...` → `ProductController@search` (routes/web.php:520) → `SearchService::search()` (name `home.search`)

Controller Logic Flow — `SearchService::search()` (`app/Services/SearchService.php`):

1. Sanitizes inputs with `strip_tags` : `query` , `category_name` , `sub_category_name` , `child_category_name` , `sub_child_category_name` ; coerces `sizes` / `brands` from comma strings into arrays; defaults minPrice 0 / maxPrice 10000.
2. Computes dealer state + `$wishlistedProductIds` for the session user.
3. Eager-loads category, subcategory, child, sub-child, brandRelation, details, sizeOnlyDetails.
4. Base `whereIn` on sellable statuses. With a query it searches `product_name` , `description` , `brief_info` , `pro_feature` , `pro_specification` , `code` , `tags` , and `whereHas` category / subcategory / child / sub-child / brand relations with `LIKE "%query%"` . An InActive product may surface if its `code` exactly equals the query (staff/quick lookup).
5. Category-level filters resolve slugs like the shop and filter by resolved ids. Price, size, brand filters behave identically.
6. Relevance boost: products whose `product_name` matches sort first (`CASE WHEN product_name LIKE ? THEN 1 ELSE 2 END`).
7. Sort mirrors the shop (COALESCE offer/price, newest, latest). Paginate 9 with `->appends(request()->except('page'))` so filter URLs persist.
8. Maps each product to set `has_offer` , `is_dealer_price_applied` , `dealer_price_calculated` , and `product_name_short` (45-char limit).

Plain-English Walkthrough: A visitor types “shirt” in the header search box and presses Enter. The search service runs `LIKE` matches across product names, descriptions, codes, and category/brand names, and puts products whose name matches first in the results. The filter sidebar works exactly like the shop and stays in the URL. If someone pastes an exact product code, even a hidden/inactive product surfaces so support staff can find it.

What the page contains: 1. Header showing the search query 2. Shop-style filter sidebar 3. Results grid using `products.list` partial 4. AJAX re-fetch on filter/sort change 5. Empty state “No products found”

Features: - **Deep full-text search across product fields** — The query runs `LIKE` searches over name, description, brief info, features, specifications, code, and tags, plus `whereHas` over category/

subcategory/child/sub-child/brand names, so a single keyword finds matches in virtually any product attribute. - **Exact-code staff lookup** — When a query exactly equals a product `code`, even an InActive product is surfaced, giving support staff a quick way to locate hidden or retired items without temporarily making them visible on the storefront. - **Relevance-ranked results** — Products whose name matches the query sort ahead of everything else (`CASE WHEN product_name LIKE ? THEN 1 ELSE 2 END`), so the most likely intended products appear first rather than being buried alphabetically or by date. - **Same filters, same engine** — Category, price, size, and brand filters reuse the shop's resolution and query logic (sellable statuses, offer-aware price ranges, COALESCE sort), so results behave exactly like the shop but scoped to the search. - **Persistent filter URLs** — Pagination uses `->appends(request()->except('page'))`, so combining a search with filters and jumping across pages keeps every part of the query intact in the URL. - **Display hygiene** — Every mapped product gets a truncated `product_name_short` (45 chars) plus `has_offer` and dealer flags, so the list view stays tidy and consistent whether the item is on offer, dealer-discounted, or regular price.

STEP 5 Product Detail (products/detail.blade.php)

Route: `GET /product/{slug}` → `ProductController@show` (routes/web.php:524) → `ProductService::getProductBySlug()` (name `home.product-detail`)

Controller Logic Flow — `ProductService::getProductBySlug()` :

1. `urldecode($slug)` ; queries with eager loads `multiImages` , `category` , `details.color` , `details.size` , `details.images` , `sizeOnlyDetails.size` ; matches `slug` OR decoded slug OR `product_name` (raw, decoded, hyphen-spaced) — `firstOrFail()` → 404 otherwise.
2. Fetches 4 related products: same `category_id` , status != `InActive` , excluding self.
3. Builds `$sizeDetailsMap` merging BOTH variant tables keyed by `size_id` : first `product->details` (each contributing `quantity_stock` , optional price/offer_price, own images), then `sizeOnlyDetails` (contributing `quantity` and, when no images exist, the main image + gallery). Duplicate sizes are summed (`quantity += ...`) and images merged uniquely.
4. Sorts the map by size name using natural string comparison (`strnatcmp`).
5. If logged in: for `Dealer Approved` computes `$dealerPrice = round((offer_price ?? price) - %, 2)` ; checks `$hasNotified` (an `Enquiry` for this product with the client's email) and `$hasOrderedBack` (an `OrderProductReceive` for this product/email with status != Resolved).
6. Returns `product, relatedProducts, sizeDetailsMap, dealerPrice, clientStatus, clientDiscount, client, hasNotified, hasOrderedBack, populars` .

Plain-English Walkthrough: A visitor clicks a product card. The service loads the product with its variants from both the `details` and `sizeOnlyDetails` tables, merges duplicate sizes, sums their stock, and sorts them naturally. The page shows the gallery with zoom, the best price (offer / dealer / original with GST), a size picker, a quantity stepper, and Add to Cart + Buy Now buttons. If the product is out of stock, Add to Cart becomes Notify Me — submitting it records an enquiry and the button flips to "requested". Related products from the same category appear below.

What the page contains: 1. Breadcrumb + title hero 2. Image gallery with hover-zoom (`#zoom-result`) 3. Price block (offer price GST-inclusive logic; dealer price when applicable) 4. “Available In” size selector (built from sizes with positive stock) 5. Quantity picker 6. Add to Cart + Buy Now buttons (AJAX add-to-cart; Buy Now adds then redirects to `/checkout`) 7. **Notify Me** form (`type='Notify'`) when out of stock — posts to `enquiry.store` 8. Tabs: Description / Features / Specifications / Package Includes / DIY Shorts / Document (document triggers the `/product/{id}/download-document` redirect route) 9. Related Products grid (`products.card`)

Features: - **Merged variant map across two tables** — The detail page merges `product->details` (colour/stock/price variants) with `sizeOnlyDetails` (standalone sizes) into one `sizeDetailsMap` keyed by `size_id` , sums duplicate quantities, and merges images — so a product stocked in both schemes shows a single coherent size picker. - **Natural size ordering** — Sizes are sorted with `strnatcmp` , so “Size 10” orders before “Size 9” correctly (not lexically), keeping the picker intuitive. - **GST-inclusive offer pricing** — The price block applies the offer price with GST math and falls back to the original price, and Dealer-approved users see a computed `$dealerPrice` rounded to 2 decimals, giving each customer class its correct price. - **Smart out-of-stock handling** — When stock is empty the page swaps Add-to-Cart for a **Notify Me** form (writes an `Enquiry`), and remembers the state via `$hasNotified` so the button shows “requested” and cannot be submitted repeatedly. - **Order-back awareness** — `$hasOrderedBack` reflects an existing unresolved `OrderProductReceive` for this product, letting the page suppress duplicate “notify when back” requests and inform the admin flow. - **Buy Now shortcut** — The Buy Now button adds the selected variant to cart and immediately redirects to `/checkout` , giving a single-click path from discovery to purchase while the normal Add to Cart keeps the shopper on the page. - **Rich content tabs** — Description, Features, Specifications, Package Includes, DIY Shorts, and a downloadable Document tab (wired to the `/product/{id}/download-document` redirect route) present the full selling story without clutter. - **Related products** — Four same-category items (excluding self) render through the shared `products.card` , cross-selling adjacent items and keeping the shopper on the product journey.

STEP 6 Offer Page (pages/offer.blade.php)

Routes: `GET /offer-promotions` → `ProductController@offer` (routes/web.php:728, name `home.offer`); legacy `GET /offer` is a static closure (routes/web.php:724-726)

Controller Logic Flow — `ContentService::getOffers()` (`app/Services/ContentService.php`):

1. `$offers = OfferPromotion::paginate(8)` (offer banners) and `$offerproducts = Product::where('status','Offer')->latest()->get()` .
2. If logged in it computes `$wishlistedProductIds` , `$notifiedProductIds` (from Enquiries by email), and dealer state.
3. Transforms each offer product exactly like the shop: `final_price_per_unit` = offer price if >0 else original; dealer price applied only when lower (`is_dealer_price_applied` , `dealer_price_calculated`).
4. Returns offers, products, wishlist/notified IDs, dealer state, categories, populars.

Plain-English Walkthrough: A visitor clicks the Offer page. Offer-promotion banners paginate on top, and below them a grid of every product flagged `status = Offer` renders using the exact same card partial as the shop. Wishlist hearts, Notify-Me state, and dealer prices are all computed the same way, so the offer page behaves identically to the rest of the store.

What the page contains: 1. Offer hero/banner section 2. “Products on Offer” grid using `products.card` 3. Same add-to-cart / wishlist behavior as the shop

Features: - **Promotion banners plus discounted catalog** — The page renders paginated `OfferPromotion` banners (8/page) above a grid of every product flagged `status = 'Offer'`, giving the marketing team both hero imagery and an evergreen discounted catalog on one page. - **Admin-controlled offer surface** — Banners and offer products are managed entirely in the admin (add/edit banners, flip a product’s status to `Offer`), so a seasonal sale can be launched without any developer involvement. - **Full shop parity** — Wishlist hearts, Notify-Me requested state, and Dealer pricing are all computed identically to the shop, so behaviour on the offer page never surprises users coming from a product card. - **Pricing consistency** — Each offer product is transformed with the same `final_price_per_unit` logic (offer > 0 ? offer : original) and the same dealer-price-only-if-lower rule used by the shop, keeping prices identical between the offer grid and the product detail page. - **Guest-friendly browsing** — The offer page is publicly viewable; interactive actions (add-to-cart, wishlist) simply invoke the standard login gating, so unauthenticated visitors can window-shop freely.

STEP 7 Wishlist (wishlist/index.blade.php)

Routes: `GET /wishlist` → `WishlistController@index` (routes/web.php:695); `POST /wishlist/add` → `store` (line 696); `POST /wishlist/remove/{id}` → `destroy` (line 697); `POST /wishlist/add-to-cart` → `addToCart` (line 699); `GET /counts` → `CartController@getCounts` (line 720)

Controller Logic Flow — `WishlistService`: 1. Requires `session('user_id')` (401/redirect gating with `show_login`). 2. `index` lists the user’s wishlist rows eager-loaded with `product`, `color`, `size`, computing per-row `display_image`, `final_price`, `applied_offer_price`, `color_name`, `size_name`. 3. `store` adds a row (unique per user + product) — returns JSON success + refreshed wishCount; guests get `login_required`. 4. `destroy` removes the row; `addToCart` copies a wishlist line into the cart then removes it.

Plain-English Walkthrough: A logged-in user taps the heart on a product card. The row saves to their wishlist and the header badge updates. On the wishlist page they see a table (or mobile cards) with image, variant, price, and two actions: quick-add, which copies the line into the cart and removes it from the wishlist, and remove, which deletes it. A guest tapping the heart gets the login modal instead.

What the page contains: 1. Breadcrumb (Home / Wishlist) 2. Desktop table: Product (image + name + color/size), Price, Action (quick-add to cart, remove) 3. Mobile card layout (image, name, variant, price, quick-add, remove) 4. Empty state: “Your Wishlist is Empty” with Browse Products CTA 5. Header badge sync via `GET /counts`

Features: - **Variant-aware wishlist rows** — Each saved line remembers the chosen colour/size and stores the computed `display_image`, `final_price`, `applied_offer_price`, `color_name`, and `size_name`, so the wishlist shows exactly what the shopper saved rather than a generic product snapshot. - **Unique per user + product** — The `store` action keeps rows unique per user/product, so repeatedly tapping the heart never creates duplicates; it simply keeps the single line (or the AJAX response refreshes the count). - **Quick-add straight to cart** — A dedicated `addToCart` action copies a wishlist line (with its variant) into the cart and removes it from the wishlist, shortening the save-then-buy journey to one click. - **Two responsive layouts** — Desktop renders a proper table (Product / Price / Actions); mobile switches to card rows with the same actions, so the management experience is usable at any screen size. - **Live header badge** — Add/remove endpoints return the refreshed `wishCount` (and the page syncs via `GET /counts`), keeping the header heart badge accurate without a reload. - **Polished empty + guest states** — Empty wishlists show “Your Wishlist is Empty” with a Browse Products CTA, and guests are routed through the shared login modal, so there is never a dead end.

STEP 8 Product Card & Quick Add (products/card.blade.php & products/list.blade.php)

Used in: Home, Shop, Search, Offer, Category, Related sections

Controller Logic Flow:

1. The card receives `$pro`, `$wishlistedProductIds`, optional `$notifiedProductIds`, and dealer/offer state from the including controller. It resolves the displayed price: offer price if lower, dealer price if `is_dealer_price_applied`, else original; highlights via `$isOfferPriceLowest` / `$isDealerPriceLowest`.
2. Renders image (`ImageHelper::getPhotoUrl`), name, brand, price/MRP, discount badges, Add to Cart, and a wishlist heart that renders filled when the id is in `$wishlistedProductIds`. Buttons call the global header JS handlers.
3. **Quick-add data** → `GET /get-quick-add-data/{id}` (routes/web.php:528, name `product.quick-add-data`) → `ProductService::getQuickAddData`: returns product id/name/image/price/offer_price plus `variants[]` built from `sizeOnlyDetails` (type `main`) and `details` (type `variant`, with color) — each with size_id, size_name, price, offer_price, stock. If no variants exist it emits a single “Standard” variant from the product’s own price/stock. This powers quick-add modals on listing pages.

Plain-English Walkthrough: Wherever products are listed, each item renders through the same card partial: image, name, brand, the best price (offer, dealer, or original) with MRP badges, a wishlist heart, and an Add to Cart button. Clicking Add to Cart opens a quick-add popup that asks the server for the product’s sizes/colours with their prices and stock. The user picks one and the chosen variant is added to the cart — no page reload. Out-of-stock items show a ribbon and switch their main action to a wishlist button.

What the card contains: 1. Product image (out-of-stock ribbon overlay) 2. Product name (truncated) 3. Price / MRP / discount badges + optional dealer price 4. Wishlist heart (filled when wishlisted) 5. Add to Cart (quick-add) button; Wishlist button for out-of-stock 6. `list.blade.php` variant: horizontal list row (image left, name, price, actions)

How to use: Any list loops `@include('frontend.products.card', ['pro'=>..., 'wishlistedProductIds'=>...])`. Quick-add UI calls the quick-add-data endpoint, renders the size/color options, and posts the chosen variant to `cart.add`.

Features: - **One partial, every listing page** — The card partial is the single rendering unit for home, shop, search, offer, category, and related grids, so price logic, badges, and buttons stay pixel-identical across the entire storefront. - **Smart price resolution with MRP badges** — The card picks the lowest applicable price (offer < dealer < original), shows discount badges against MRP, and flags which price won via `$isOfferPriceLowest` / `$isDealerPriceLowest`, making sale and dealer deals visually obvious. - **Out-of-stock ribbon overlay** — Products with no stock show an overlay ribbon and switch their primary action to a wishlist button, so visitors immediately understand availability without opening the page. - **Wishlist state on every card** — Hearts render filled for ids in `$wishlistedProductIds`, and toggling posts through the shared header JS so the wishlist count and heart state stay in sync everywhere. - **Quick-add modal without a page load** — Clicking Add to Cart on a listing opens a size/colour picker fed by `GET /get-quick-add-data/{id}`, which returns the merged variant list (`sizeOnlyDetails` as “main”, `details` as “variant”) with per-variant price/offer/stock; products with no variants get a single “Standard” option, so the modal always has something to add. - **Horizontal list variant for search** — `products/list.blade.php` lays the same data out as an image-left row for search results, giving a denser presentation while reusing the identical price and action logic.

PART 3 — CART, COUPONS, DIY POINTS

STEP 1 Cart Page (cart/index.blade.php)

Routes: `POST /add-to-cart` (`cart.add` , web.php:658), `GET /cart` (`cart.show` , web.php:660), `POST /cart/update-quantity` (web.php:662), `POST /delete-cart-item` (web.php:663), `POST /cart/move-to-wishlist` (web.php:702), `GET /cart/get-totals` (web.php:782), `GET /counts` (web.php:720) → `CartController` → `CartService` (750 lines)

Controller Logic Flow — `CartService::addToCart` : 1. Forgets the user's DIY-points session (`diy_points_{userId}`) so a changed cart invalidates stale points. Requires login (else `login_required`). 2. Validates `product_id` + `quantity >= 1` . If no `selected_size` , auto-picks the lowest-priced in-stock variant (checks `details` first, then `sizeOnlyDetails`). 3. Finds the line matching `user_id + product_id + size_id + color_id` ; increments `quantity` or creates a new line. Recomputes `cart_count` (sum of quantity) and stores `session('cart_count')` . 4. Returns `{success, cart_count}` and, when `buy_now` present, `redirect_url` → `/checkout` .

`CartService::showCart` : 1. Loads dealer state (status `Dealer Approved` → `clientDiscountPercentage`), cart items eager-loaded with `product.details.images` , `product.sizeOnlyDetails` , `product` , `color` , `size` , plus wallet balance and the session DIY points. 2. Reads coupon sessions `coupon_{userId}_coupon` / `coupon_{userId}_pin` ; empty cart forgets both. A session coupon re-validates against the DB: active unexpired `Coupon` OR `Unused Pin` owned by the user; else forgets and nulls. 3. Per-item pricing: resolves detail/sizeOnly price + offer, clamps quantity to max stock, computes `priceAfterOffer` (offer>0 ? offer : original), applies dealer % only when lower, splits GST by `product_gst` rate (`base = incl/(1+rate)` , `gst = incl-base`), accumulates `subtotalBeforeClientDiscount` , `totalClientDiscountApplied` , `totalGstAmount` . 4. Extra discount from `ExtraDiscount::first()` when subtotal >= `total_cart_value` (`cart_discount%`). 5. `cartTotal = max(0, subtotal - couponDiscount)` ; `cartTotalInclGst = cartTotal + gst` . 6. DIY potential: `Referal::first()` settings (`max_diy_points` 100, `diy_points_divisor` 60, `wallet_cap` 333). `canEarnPoints` only when no DIY used, no coupon, no extra discount, no dealer discount, and wallet < cap. 7. Returns cart items (with computed dynamic attributes), coupon details, subtotals, GST, dealer flags, wallet balance, DIY used, potential points.

Other actions: `updateQuantity` (clamps to stock, recomputes summary JSON incl. DIY), `deleteItem` (recompute via `Cart::with('product')`), `moveToWishlist` (insert matching `Wishlist` row if absent then delete cart line), `getTotals` (full summary JSON), `getCounts` (cartCount + wishCount for header badges).

Plain-English Walkthrough: A logged-in user opens `/cart` . The page shows each line with image, variant, SKU, a quantity stepper, price (base + GST, with a Dealer badge when applicable), and subtotal. Changing the quantity re-fetches the whole summary sidebar via AJAX. The user can type a

coupon code, enter DIY points, or move a line to the wishlist instead of deleting it. The summary lists subtotal, dealer discount, extra discount, coupon discount, GST, DIY points applied, and the final total; “Proceed to Checkout” continues the flow.

Worked Example (with real numbers): Say a user adds 2 shirts to the cart. Shirt A has an offer price of ₹700 (MRP ₹1,000, GST 18%), Shirt B is regular price ₹800 (MRP ₹1,000, GST 18%). No dealer discount, no coupon, no DIY points yet. 1. Item A: offer applies → base line = ₹700. 2. Item B: no offer → base line = ₹800. 3. Subtotal before discount = ₹700 + ₹800 = **₹1,500**. 4. GST split on each line (base = incl ÷ 1.18): A → base ₹593.22 + GST ₹106.78; B → base ₹677.97 + GST ₹122.03. Total GST = **₹228.81**. 5. Extra discount: the store rule is “₹2,000 → 5%”, but ₹1,500 < ₹2,000 → **₹0**. 6. Coupon: none applied → **₹0**. DIY points: none → **₹0**. 7. Cart total (excl GST) = **₹1,500**; total incl GST = **₹1,728.81**. Now suppose the user adds a third shirt (₹700): the item subtotal becomes ₹2,200, which crosses the ₹2,000 threshold → a 5% extra discount of **₹110** is automatically deducted and shown on the summary.

What the page contains: 1. Breadcrumb + “Cart” header 2. Desktop table: Product (image + name + variant), SKU code, Quantity stepper (clamped to stock), Price (base + GST + Dealer badge), Subtotal, Actions (move-to-wishlist heart, trash) 3. Mobile card layout (image, name, variant, price, quantity controls, actions) 4. Coupon apply box + applied-coupon panel with remove button 5. Order summary panel (Subtotal, Dealer Discount, Extra Discount, Coupon Discount, GST, DIY Points Applied, Total) recalculated via AJAX 6. “Proceed to checkout” → `home.checkout` 7. Empty cart graphic + message; JS `checkCartEmpty()` toggles after deletions

Features: - **Stock-clamped quantity controls** — The quantity stepper can never exceed available stock (verified against the merged variant tables), and `updateQuantity` enforces the same clamp server-side, so the cart can never hold more than is purchasable. - **Automatic variant selection** — Adding a product without picking a size auto-selects the lowest-priced in-stock variant (checking `details` then `sizeOnlyDetails`), so one-click add-to-cart from a card always produces a valid, priceable cart line. - **Full pricing engine in the service layer** — `CartService` centralises offer/dealer/GST/extra-discount/coupon/DIY math (including GST split as `base = incl/(1+rate)`), guaranteeing the cart, checkout, order, and invoice all compute money the same way. - **Live re-validation of coupons** — Any stored coupon/pin is re-checked against the DB (active, unexpired, `Unused` pin owned by the user) on every cart load and silently dropped when invalid, so expired deals never survive into checkout. - **Move-to-wishlist instead of delete** — A cart line can be moved to the wishlist in one click (creating the wishlist row when absent and removing the cart line), turning an abandoned item into a saved one. - **AJAX-refreshed summary** — Quantity changes and item deletions re-fetch the full order summary JSON (subtotal, all discounts, GST, DIY points, total) so the sidebar always reflects the current cart without a reload. - **DIY-points invalidation on cart change** — Adding an item forgets the user’s session DIY points, forcing a re-decision after the cart changes — preventing stale point amounts being applied to a different total.

STEP 2 Coupon Engine (admin coupon + generated PIN)

Routes: `POST /coupon/apply` (`coupon.apply` , web.php:670), `POST /coupon/remove` (web.php:671), `POST /coupon/create` (web.php:669), `GET /user-coupon` (web.php:736), `POST /user/coupon/add` (web.php:737), `GET /user/coupon/deposit` (web.php:739) → `CouponController` → `CouponService`

Controller Logic Flow — `CouponService::apply` : 1. Requires login; requires a non-empty cart (sums `offer_price>0 ? offer_price : price` per item as `cartSubtotal`). 2. First checks `Coupon` (active + not expired): existing `CouponUsage` for that name → "already used"; `min_amount` not met → minimum-total rejection. Otherwise stores session `coupon_{userId}_coupon` = `{source:'coupon', coupon_name, discount_amount, coupon_id, code}` . 3. Falls back to `Pin` (user-generated coupon): active `Pin` matching the code, owned by the user, status `Unused` → stores `coupon_{userId}_pin` = `{source:'pin', coupon_name, discount_amount, pin_id, code}` . 4. If neither → "Invalid, expired or unauthorized coupon."

Remove flow: `removeCoupon` forgets both session keys.

Generate flow (`CouponService::addCoupon`): requires a Wallet; `amount` must be <= wallet balance; in a DB transaction it decrements the wallet, generates `transaction_no` (12 hex chars), `coupon_code` (7 random A-Z0-9), `coupon_name` (9 random), sets `generated_date = today` , `status = Unused` , and inserts into the `pins` table via raw SQL. Rolls back on failure.

Usage recording: at order time (paid orders only) `CheckoutService::placeOrder` writes `CouponUsage::firstOrCreate` per coupon/pin used.

Plain-English Walkthrough: A user types a coupon code in the cart and hits Apply. The service first checks the admin Coupon table — is it active, unexpired, not already used by this user, and does the cart meet the minimum amount? If that fails, it tries a user-generated PIN that the user owns and hasn't used yet. On success the discount is stored in the user's session and shown on the summary. A user can also generate their own PIN coupon from wallet balance: the amount is deducted from their wallet and a code is created. The coupon is only marked as used when a paid order actually completes.

Worked Example (with real numbers): 1. **Admin coupon applies.** Cart subtotal = **₹2,000**. The user enters code `SAVE10` . The service finds an active admin Coupon `SAVE10` worth **₹150** with `min_amount = ₹999` : the cart (₹2,000 ≥ ₹999) qualifies, the user has not used it before, and it is not expired → the discount **₹150** is stored in the session and the new total is **₹1,850**. 2. **Admin coupon rejected → PIN tried.** If `SAVE10` were already used or expired, the service falls back to the user's own PINs; a matching `Unused` PIN worth **₹200** would apply instead → total **₹1,800**. 3. **Nothing valid.** Both checks fail → "Invalid, expired or unauthorized coupon", total unchanged. 4. **Generating a PIN coupon.** The user goes to My Coupons and enters **₹250** (wallet balance must be ≥ ₹250). The wallet is debited ₹250 and a new coupon code (e.g. `AB3K9XZ` , value ₹250) is created with `status = Unused` . 5. **Consumption timing.** None of the coupons above are marked used at apply time — `CouponUsage` is only written when a paid order completes, so a cancelled Razorpay payment does NOT burn the coupon and the user can retry with it.

Features: - **Two coupon sources, one discount slot** — The engine accepts either an admin-created `Coupon` (single-use-per-user, optional `min_amount` threshold) or a user-generated `Pin`, and stores whichever applies into the same session discount slot, so checkout treats both identically. - **Single-use enforcement via usage table** — Applying an already-used admin coupon is rejected because `CouponUsage` records consumption at order time, and PINs must have `status = Unused`; nothing can be applied twice. - **Session-scoped discount state** — Applied coupons live in per-user session keys (`coupon_{userId}_coupon` / `_pin`) and are re-validated against the database on every cart/checkout load, so an expired or spent code silently falls off instead of erroring. - **Wallet-funded coupon generation** — `addCoupon` debits the user's wallet in a transaction, generates a 7-char code + 9-char name + 12-hex transaction number, and writes a `Pin` via raw SQL — all or nothing, so users can convert balance into transferable discount codes. - **Only paid orders consume coupons** — `CouponUsage` is written only for fully-paid orders, so a cancelled/failed payment does not burn the coupon and the user can retry with it. - **Clear rejection messages** — The apply endpoint distinguishes "already used", "minimum cart not met", and "invalid/expired/unauthorized", so users understand exactly why a code was rejected.

STEP 3 DIY Points / Wallet usage

Routes: `POST /diy-points/apply` (`diy.points.apply`, web.php:673), `POST /diy-points/remove` (`diy.points.remove`, web.php:674) → `CouponController@applyDIYPoints/removeDIYPoints` → `DiyPointsService`

Controller Logic Flow — `DiyPointsService`: 1. `applyPoints` validates `diy_points >= 1`, requires login, loads the wallet, rejects if the requested points exceed the balance, then stores `session(['diy_points_{userId}' => $pointsToUse])`. All cart/checkout totals read this session value. 2. `removePoints` forgets it; `calculateDiscount` just reads it back. 3. Everywhere the value is used it is clamped: `diyPointsUsed = min(session_value, subtotalBeforeClientDiscount)` (CartService.updateQuantity, getTotals, CheckoutService). 4. `canEarnPoints` rules (from Referal settings) block earning while any discount (coupon/extra/dealer) is active or points are being used — points are earned only on "clean" fully-paid orders. 5. At order placement, `CheckoutService::placeOrder` debits the wallet (`payment_method='diypoint_usage'`) and awards earned points (`payment_method='diypoint'`) for eligible orders (see Part 4).

Plain-English Walkthrough: A user with wallet balance enters DIY points at checkout and clicks Apply. The service verifies the points don't exceed their balance and stores the amount in the session. Every total from then on reads that value and clamps it to the current cart subtotal. If the user changes the cart afterwards, the points are cleared so they must re-decide. Points are only earned on clean, fully-paid orders that used no discounts, and the amount earned is capped by the site settings.

Worked Example (with real numbers): 1. **Applying points:** The user's wallet balance is **₹500**. Their cart subtotal (excl GST) is **₹1,200** with no discounts active. 2. They enter **300 DIY points** and click Apply → 300 ≤ 500 (balance OK) → 300 is stored in the session. 3. Every total now reads `diyPointsUsed = min(300, 1200)` = **₹300** off → the payable amount drops by ₹300. If the cart later

shrank to ₹200, the clamp would make it `min(300, 200)` = **₹200** so they can never "over-apply". 4. **Points cleared on change:** If the user adds another item to the cart, `addToCart` forgets the DIY session value → they must re-apply points against the new total. 5. **Earning points:** After a clean, fully-paid order with subtotal (excl GST) of **₹3,000**, and site settings `diy_points_divisor = 60`, `max_diy_points = 100`, wallet cap **₹333**: earned = `floor(3000/60)` = **50 points** (50 < 100 cap, and wallet stays under 333). No points are earned if any coupon/extra/dealer discount was used or if points were redeemed on this order.

Features: - **Wallet balance as spendable points** — DIY points ARE the wallet balance: `applyPoints` writes the requested amount to the per-user session and every totals calculation reads and clamps it, turning the stored balance into an instant checkout discount. - **Server-side balance check** — Applying points validates the wallet actually covers the request and rejects over-application, so the session can never promise more than the user owns. - **Clamped application everywhere** — Every consumer (`updateQuantity`, `getTotals`, `placeOrder`) recalculates `diyPointsUsed = min(session_value, subtotalBeforeClientDiscount)`, so the applied amount never exceeds the current cart subtotal even if the cart shrank since the points were set. - **Earning is gated on "clean" orders** — `canEarnPoints` returns false while a coupon, extra discount, or dealer discount is in play, or when points are being used, or when the wallet is at cap — so points are only earned on simple, fully-paid purchases (preventing discount stacking abuse). - **Single transaction writes both sides** — Order placement debits usage (`diypoint_usage`) and credits earned points (`diypoint`) as separate labeled wallet transactions, giving the transaction ledger a complete, auditable points history. - **Points invalidate on cart change** — `addToCart` forgets the DIY session key, forcing the user to re-apply points after any cart modification so the discount always matches the current total.

STEP 4 Extra Discount & Free Shipping logic

Models: `ExtraDiscount` (single row: `total_cart_value`, `cart_discount` %), `FreeShipping` (single row: `shipping_amount`, `shipping_text`)

How it works:

1. **Extra discount** applies in `CartService::showCart/updateQuantity/getTotals`, `CheckoutService::getCheckoutData`, and `CheckoutService::placeOrder`: whenever the (item-discounted) subtotal meets `total_cart_value`, a flat `%` is subtracted and surfaced as `extraDiscountDetails` / `extraDiscountAmount`, and stored on the order (`extra_discount_amount`).
2. **Free shipping** is evaluated in `getCheckoutData` (final grand total >= `shipping_amount` → `isFreeShipping=true`) and again in `placeOrder` (same threshold zeroes `$shippingCost`). The checkout view shows "Free Delivery" and the JS `calculateShipping` bails to free when `isFreeShippingInitial`.
3. Shipping cost otherwise comes from the address's `stateRelation->shipping_cost` ; `CheckoutService::getShippingFee($addressId)` is exposed via `GET /get-shipping-fee/{addressId}` (web.php:741) so checkout fetches it live per selected address.

Plain-English Walkthrough: If the store has an Extra Discount rule (for example “spend ₹2000 get 5% off”), the cart automatically applies it once the item total crosses the threshold; the same check runs again at checkout and when the order is saved, and the amount is stored on the order so every page agrees. Free shipping works the same way: once the final total meets the threshold, shipping is zeroed everywhere and the checkout shows “Free Delivery”. When free shipping doesn’t apply, the shipping fee comes from the delivery state, and the checkout refetches it live when the user changes the delivery address.

Worked Example (with real numbers): Admin config: `ExtraDiscount` → `total_cart_value = ₹2,000`, `cart_discount = 5%`. `FreeShipping` → `shipping_amount = ₹500`. 1. **Cart A:** items total (after offers/dealer) = **₹2,400** → extra discount 5% = **₹120** → subtotal ₹2,280. Final total ₹2,280 ≥ ₹500 → **free shipping** (₹0), checkout shows “Free Delivery”. 2. **Cart B:** items total = **₹1,500** (no extra discount since < ₹2,000). State shipping = **₹60**. Final total ₹1,560 ≥ ₹500 → **free shipping still applies** (₹0 charged). 3. **Cart C:** items total = **₹300**. State shipping = **₹60**. Final total ₹360 < ₹500 → **₹60 shipping is charged**. 4. The same three checks run identically inside `placeOrder`, so the amount saved on the cart is exactly the amount stored on the order (`extra_discount_amount`) and shown on the success page and invoice.

How to use: Admin edits the single ExtraDiscount and FreeShipping rows; the storefront automatically applies thresholds at every pricing point (cart, checkout, order record, success/failed pages).

Features: - **Automatic threshold discount** — When the item-discounted cart value meets the admin-configured `total_cart_value`, a percentage `cart_discount` is applied automatically across the cart, checkout, and order placement — a built-in “spend X, save Y%” promotion with zero code changes. - **Consistent across every money view** — The exact same extra-discount computation runs in `showCart`, `updateQuantity`, `getTotals`, `getCheckoutData`, and `placeOrder`, and the final amount is persisted on the order (`extra_discount_amount`) so the success page and invoice show what was actually applied. - **Free-shipping threshold as a single row** — One `FreeShipping` row defines the threshold; both checkout data and order placement evaluate the same condition, and checkout displays “Free Delivery” plus an initial flag (`isFreeShippingInitial`) the JS respects before any per-state shipping fetch. - **Per-state shipping costs** — When not free, shipping is derived from the delivery state’s `shipping_cost`, and the checkout re-fetches it live (`GET /get-shipping-fee/{addressId}`) whenever the user switches the delivery address, so the quote always matches the chosen state. - **Deterministic precedence** — Extra discount is applied to the item-discounted subtotal BEFORE coupons and DIY points in the cascade, keeping the discount stack predictable and preventing double-discount edge cases.

PART 4 — CHECKOUT & ORDERS

STEP 1 Checkout Page (checkout/index.blade.php)

Route: `GET /checkout` → `CheckoutController@checkout` (web.php:677) → `CheckoutService::getCheckoutData()` (name `home.checkout`) **Gate:** session login (`show_login`)

Controller Logic Flow — `CheckoutService::getCheckoutData()`: 1. Loads dealer state; loads all shipping addresses (with city/state/mandal relations) oldest-first, splitting into `$addresses` (type shipping OR empty) and `$billingAddresses` (type billing). 2. Loads cart items (eager: details.images, sizeOnlyDetails, color, size). Empty cart → forgets coupon sessions. 3. Per-item pricing identical to cart: detail/sizeOnly price+offer, clamp qty to max stock, `priceAfterOffer`, dealer discount (only when lower), GST split, accumulating subtotals. Sets dynamic per-item attributes (`final_price` , `item_total_incl_gst` , `gst_amount`). 4. Extra discount from `ExtraDiscount` (threshold on grand total incl GST) → `grandTotalAfterExtraDiscount` . 5. Coupon/pin re-validation exactly like the cart → `discountAmount` + refreshed `couponDetails` . 6. `finalGrandTotal = max(0, grandTotalAfterExtraDiscount - discountAmount)` ; dealer subtractions; DIY recompute (`cartSubtotalAfterDIY + totalGST - extra - coupon`). 7. Free-shipping check on finalGrandTotal; wallet balance; potential DIY points with a special rule: if the client was `referred_by` and has zero prior orders, `canEarnPoints = false` (first order of a referred user is excluded).

Plain-English Walkthrough: A logged-in user with items in the cart clicks Proceed to Checkout. The page loads their saved billing and shipping addresses, an itemized summary, and payment options. They pick (or add) addresses using dependent state→district→mandal selects, can choose Store Pickup or “Same as Billing”, type special instructions, and pick Razorpay, Wallet, or COD. Switching the delivery address refreshes the shipping fee live. Submitting the form hands everything to the order-placement logic.

Worked Example (with real numbers): Cart: 1 item **₹800** (GST 18%), no dealer discount. A coupon of **₹100** is applied. The user enters **200 DIY points**. The chosen state’s shipping is **₹60**; the free-shipping threshold is **₹500**. 1. Item line: ₹800 incl GST → base **₹677.97** + GST **₹122.03**. 2. Grand total incl GST = **₹944**. Extra discount threshold (₹2,000) not met → ₹0. 3. Coupon ₹100 → ₹844. DIY recompute: `cartSubtotalAfterDIY = 800 − min(200, 800) = ₹600` , plus GST ₹122.03, minus coupon ₹100 → **₹622.03**. 4. Free shipping: ₹622.03 ≥ ₹500 → shipping **₹0**, page shows “Free Delivery”. 5. Wallet balance shown (e.g. **₹900**) for the wallet payment option; if the user had been referred and this were their first order, `canEarnPoints` would read false and the “potential points” line would be hidden. 6. User pays ₹622.03 via Razorpay → popup → success handler submits the payment id to `order.store` .

What the page contains: 1. Breadcrumb + “Checkout” header + admin `checkout_notice` alert 2. Billing Address panel (saved-address radios + “Add New” drawer) 3. Shipping Address panel (saved-

address radios with delivery estimate, “Same as Billing” checkbox, **Store Pickup** option from `$setting->store_address`) 4. Specific Instructions textarea 5. “Add New Address” drawers with dependent State→District→Mandal→PIN selects (via `get-cities` / `get-mandals` / `get-pin-code` routes) 6. Order summary sidebar (products + qty/size, subtotal, shipping fee, coupon, extra discount, DIY points, total) 7. Payment Method radios: Razorpay / Wallet / Cash on Delivery 8. Razorpay popup integration (success handler posts `razorpay_payment_id` to `order.store`); wallet-confirmation against `$userWallet->balance`

Features: - **Split billing/shipping management** — Addresses are loaded once and split by `address_type` into billing and shipping panels, letting the user pick saved addresses or add new ones inline without leaving the page. - **Store Pickup as a first-class option** — A pickup radio uses the admin-configured `$setting->store_address`, and order placement treats `Pickup` as a paid zero-shipping path, giving a real click-and-collect channel. - **Same-as-billing shortcut** — The “Same as Billing” checkbox collapses the shipping step, and the backend mirrors the billing id when set, reducing friction for the common case. - **Dependent address dropdowns** — New-address drawers chain State → District → Mandal → PIN via the `get-cities` / `get-mandals` / `get-pin-code` endpoints, auto-filling the PIN and validating the Indian format before submit. - **Live shipping quotes** — Switching the delivery address re-fetches `GET /get-shipping-fee/{addressId}` so the sidebar’s shipping line updates immediately, and free shipping short-circuits the fetch. - **Payment method coverage** — Razorpay (popup, with the payment id posted to `order.store` on success), Wallet (with a balance check against `$userWallet->balance`), and COD are offered; cancelled Razorpay flows land on the failed page with the cart preserved. - **First-order referral exclusion** — `canEarnPoints` is forced false for a referred client’s first-ever order (referred_by set + zero prior orders), matching the reward rules so signup-referral users don’t double-earn on their first purchase.

STEP 2 Place Order (CheckoutService.placeOrder)

Route: `POST /checkout/order` (`order.store`, web.php:682) → `CheckoutController@store` → `CheckoutService::placeOrder($request, $userId)`

Step-by-step logic:

1. **Load the client and set the dealer state.** `placeOrder` loads the logged-in Client and checks whether their `client_status` is `Dealer Approved`; if so it records the discount percentage to use. It then reads the chosen billing and shipping address ids from the request. If the user ticked “Same as Billing”, the shipping id is set equal to the billing id. If no shipping id was provided, the request fails with an error; if no billing id was provided, the shipping id is used as the fallback. Unless the order is Store Pickup, the shipping address is loaded together with its state and mandal relations.

2. **Load the cart and recompute every price from the database.** The user’s cart is loaded; an empty cart aborts with an error. Critically, prices are NOT taken from what the browser sent — for each line the service re-reads the actual `ProductDetail` / `SizeOnlyDetail` rows, applies the offer price when one exists, applies the dealer discount when it is lower, and accumulates

`totalItemsInclGST` (everything the customer pays including GST) and `totalGSTAccrued` (the GST portion alone).

3. **Apply the extra discount and coupon discounts.** The single-row `ExtraDiscount` config is checked: when `totalItemsInclGST` meets the configured `total_cart_value`, the flat percentage is deducted. The coupon/pin discount stored in the session earlier (from the cart or checkout) is read next. The running total becomes `totalAfterAllDiscountsInclGST = max(0, totalItemsInclGST − extraDiscount − couponDiscount)`, so discounts can never drive the total below zero.
4. **Compute the shipping cost.** For delivery orders, shipping comes from the delivery state's `shipping_cost` (`address->stateRelation->shipping_cost`). It is forced to zero in exactly two cases: when the free-shipping threshold is met, or when the payment method is Store Pickup.
5. **Apply the dealer discount at the order level.** If the client is Dealer Approved, the dealer percentage is also applied across the whole order (not just per item), so the final amount reflects the wholesale rate consistently.
6. **Redeem DIY points from the wallet.** The effective DIY points are read from the session (falling back to the request value). If the value is greater than zero and the wallet balance covers it, the applied amount is clamped to the cart subtotal excluding GST — `diyPointsUsed = min(points, subtotalExclGST)`. The wallet is decremented, a `WalletTransaction` with `payment_method='diypoint_usage'` and id `DIY-USE-...` is written, and all totals are recomputed to reflect the redemption.
7. **Compute the final amount and the order type.** `finalAmountForOrder = totalAfterAllDiscounts + shippingCost`. The `order_type` is set to `pickup` or `delivery` depending on whether Store Pickup was chosen.
8. **Run the payment branch (see Step 3 below).** Depending on the chosen method (Razorpay with or without a payment id, Wallet, COD, or Pickup) the service sets `paidAmount`, `paymentStatus`, `paymentId`, `orderStatus`, `processingAt`, and `isPaid`. This single decision controls everything downstream, because rewards and coupon usage only execute when the order is paid.
9. **Record coupon usage for paid orders.** Only when the order is paid does `CouponUsage::firstOrCreate` run for the coupon and/or pin names used — so a code consumed on one paid order can never be applied to a second order.
10. **Create the `Order` record.** The order row is inserted with the resolved values: address ids are nulled for pickup orders (there is no delivery address), `sale_by='ByClient'`, the coupon and extra discount amounts are stored so every history page can reproduce the exact pricing, plus `order_instructions`, the order status, `processing_at` (when the order entered processing), and `is_paid`. If DIY points were redeemed, the DIY wallet transaction is linked to this order id so the ledger can always be traced back to the order.
11. **Write the `OrderList` line items.** One row per cart item is inserted with `order_id`, `product_id`, the product's `hsn_code` (needed for GST reporting), `quantity`, the final per-unit price including

any dealer discount, and the chosen `size_id` / `color_id`. This snapshot freezes exactly what was sold and at what price, even if the product or price changes later.

12. **Decrement stock.** While the order status is one of `Processing`, `Order received`, or `Pending`, stock is reduced on the exact variant rows that were priced — `ProductDetail.quantity_stock` or `SizeOnlyDetail.quantity`. This is what keeps inventory in sync with actual sales.

13. **Send the order confirmation email.** `OrderSuccessMail` is dispatched to the client's email through `BrevoMailService`, wrapped in try/catch so a mail failure is logged but never aborts the order — the order stands regardless.

14. **Award the referral reward (paid + referred orders only).** If the order is paid and the client was referred by someone, the service first checks the referrer hasn't already been rewarded for this client. It then validates the client registered within 7 days and the order total meets `min_order_value_for_referal`, before awarding a random amount from the configured range, capped by the wallet cap, as a `WalletTransaction` with `payment_method='Referralbonus'` and id `REF-PT-...`.

15. **Credit earned DIY points (paid, clean orders only).** For orders that are paid in full, are not offline or replacement orders, are not a referred client's first order, and used NO discounts (coupon / extra / dealer / DIY), the earned points are `floor(round(subtotalExclGST) / diy_points_divisor)`, capped by `max_diy_points` and the wallet cap, and written as `payment_method='diypoint'` with id `DIY-PT-...`.

16. **Finalize the order.** The cart is cleared, the coupon/pin/DIY/shipping session keys are forgotten, and the method returns `{success, order_id, redirect → order.success}` so the controller can send the user to the confirmation page.

Plain-English Walkthrough: When the user submits checkout, the server recalculates every price directly from the database — nothing the page sent is trusted. It resolves the addresses (same-as-billing, pickup, or fallbacks), applies offer/dealer/extra/coupon/DIY discounts in order, computes shipping or zeroes it, then branches on the payment method: a Razorpay id means paid, wallet deducts the balance, pickup counts as paid, COD stays pending. It creates the order and its item lines, reduces stock, emails the customer, and — only for paid orders — records coupon usage, awards referral rewards, and credits earned DIY points. Finally it empties the cart and sends the user to the success page.

Worked Example (with real numbers): One shirt, offer price **₹1,000**, GST **18%**. The user is Dealer Approved (**10% off**). A coupon of **₹100** is applied. The user redeems **200 DIY points**. The state's shipping cost is **₹70** and the free-shipping threshold is **₹1,000** (not met). Payment is Razorpay (completed). 1. Client loaded; dealer discount 10% recorded. Addresses resolved (delivery order). 2. Cart = 1 shirt. Recomputed from DB: offer ₹1,000 → dealer 10% → **₹900** per unit incl GST. `totalItemsInclGST = ₹900`; `totalGSTAccrued = 900 − (900 ÷ 1.18) = ₹137.29`. 3. Extra discount: ₹900 < ₹2,000 threshold → ₹0. Coupon ₹100 → `totalAfterAllDiscountsInclGST = max(0, 900 − 100) = ₹800`. 4. Shipping: ₹70 (threshold ₹1,000 not met) → **₹70**. 5. Dealer discount already reflected per unit in step 2. 6. DIY: 200 points requested, wallet covers it → `diyPointsUsed = min(200, 800 ÷ 1.18 = ₹677.97) = ₹200`. Wallet debited ₹200; `DIY-USE-...` transaction written.

Total becomes ₹800 − ₹200 = **₹600**. 7. `finalAmountForOrder = ₹600 + ₹70 = ₹670`; `order_type = delivery`. 8. Payment branch: Razorpay id present → `paidAmount = ₹670`, status `Completed`, id = razorpay id, order status `Order received`, `processing_at = now`, `is_paid = 1`. 9. `CouponUsage` recorded for the coupon. 10. `Order` created: address ids kept, `sale_by = ByClient`, `discount_amount = ₹100`, `is_paid = 1`; DIY transaction linked to the order. 11. `OrderList`: 1 row — product id, hsn_code, qty 1, unit price ₹900 (dealer, incl GST), size/color ids. 12. Stock: `ProductDetail.quantity_stock` reduced by 1. 13. `OrderSuccessMail` sent. 14. Referral reward: not a referred client → skipped. 15. Earned DIY points: **₹0** — because a coupon was used, this is not a "clean" order, so no points are earned (illustrating the no-discount rule). 16. Cart cleared, sessions forgotten, redirect to success.

How to use: Runs automatically on checkout form submit. Failures return `error` back to checkout or redirect to the failed page.

Features: - **Server-side recomputation (never trust the cart client)** — Every price, GST split, dealer discount, and total is recomputed directly from `ProductDetail` / `SizeOnlyDetail` lookups inside `placeOrder`, so a tampered cart or stale client total can never determine what the customer is charged. - **Address resolution rules** — The order intelligently resolves `same_as_billing` to the billing id, falls back to shipping id when no billing is chosen, and errors cleanly when no shipping address exists — while pickup nulls the address ids entirely. - **Single transactional flow for payments and rewards** — The order, order lists, stock decrement, coupon usage, wallet debits, and point credits all happen in one orchestrated flow, with rewards (referral, earned DIY) gated on `isPaid` so nothing is granted for unpaid orders. - **Order status reflection of payment** — COD/pending orders record `is_paid=0` with a `COD-{rand}` id; paid orders record `Completed` with a real payment id and `Order received` + `processing_at`, giving the client area a truthful order status pipeline. - **Stock integrity** — Stock is decremented only while the order status is `Processing / Order received / Pending`, against the exact variant rows (`ProductDetail.quantity_stock` or `SizeOnlyDetail.quantity`) that were priced. - **Automated customer communication** — `OrderSuccessMail` is dispatched through `BrevoMailService` inside try/catch, so a mail failure never aborts the order — the order stands and the failure is logged. - **Fair referral rewards** — Referral bonus applies only when the referrer hasn't already been rewarded for this client, the client registered within 7 days, and the order meets `min_order_value_for_referal`; the award is a randomized range capped by the wallet cap, preventing exploitation. - **Earned-point precision** — Earned DIY points use `floor(round(subtotalExclGST)/divisor)` capped by `max_diy_points` and the wallet cap, and are skipped whenever any discount was used — keeping the loyalty currency from stacking with promotions.

STEP 3 Payment Paths (Razorpay / Wallet / COD / Pickup)

How it works (inside the `placeOrder` payment branch):

1. **Razorpay/Card/NetBanking/PayLater**: no `razorpay_payment_id` → creates a **Cancelled** order (`payment_status='Cancelled'`, `paid_amount=0`, id `CANCELLED-{rand}`) with all `OrderList`

rows, stores `session(['failed_order_id'=>...])`, redirects to `order.failed`. If the payment id IS present → `paidAmount = finalAmount`, status `Completed`, id = razorpay id, order status `Order received`, `processing_at = now()`.

2. **Wallet**: requires wallet balance >= `finalAmount`; decrements wallet, `paidAmount = finalAmount`, status `Completed`, id `WAL-{rand}`, status `Order received`.
3. **Pickup**: treated as paid (`Completed`, id `PICKUP-{rand}`, `Order received`), shipping zeroed.
4. **COD / anything else**: status `Pending`, id `COD-{rand}`, order status `Order received`, `is_paid=0`.
5. `isPaid` = status `Completed` → 1. Coupon usage and all rewards (referral, earned DIY) run only when `isPaid`.
6. The checkout page creates the Razorpay order via `CheckoutController@createRazorpayOrder` (`POST /razorpay/create-order`, web.php:767): sums offer/price × qty, calls the Razorpay SDK, returns `{order_id, amount}` in paise. The page JS opens the popup and submits the form with `razorpay_payment_id` on success.

Plain-English Walkthrough: On checkout the user picks a payment method. Razorpay opens a popup for a pre-created order: completing it submits the payment id and the order is marked paid; cancelling it records a Cancelled order and sends the user to the failed page with the cart intact. Wallet checks the balance, deducts it, and marks the order paid. Pickup marks the order paid with zero shipping. COD records the order as payment-pending. All rewards and coupon usage only happen when the order is paid.

Worked Example (with real numbers): The same order needs **₹670** (`finalAmountForOrder`). Here is what each method does with it: 1. **Razorpay (completed):** `paidAmount = ₹670`, `paymentStatus = Completed`, `paymentId` = the Razorpay id, order status `Order received`, `is_paid = 1`. Coupon usage, referral reward, and earned DIY points all run. 2. **Razorpay (cancelled):** no `razorpay_payment_id` → a `Cancelled` order is created (`paid_amount = 0`, id `CANCELLED-xxx`) with its item rows, `failed_order_id` is stored in the session, and the user is redirected to `/order/failed`. The cart is untouched, so they can retry. 3. **Wallet:** the wallet balance (say **₹900**) must be ≥ ₹670 → it is decremented to **₹230**, `paidAmount = ₹670`, status `Completed`, id `WAL-xxx`, `is_paid = 1`. 4. **Store Pickup:** treated as paid (`Completed`, id `PICKUP-xxx`, `Order received`), and shipping is zeroed — the ₹70 state shipping from the delivery example is never charged. 5. **COD:** `paidAmount = 0`, id `COD-xxx`, status `Pending`, `is_paid = 0` — the order is recorded but earns no rewards until the payment is settled.

How to use: Select a method on checkout. Razorpay opens a popup; complete or cancel it. Wallet/COD/Pickup submit directly. Cancelled payments land on the Failed page with a recorded cancelled order.

Features: - **Razorpay popup integration** — Checkout pre-creates a Razorpay order (`POST /razorpay/create-order`) returning an order id + paise amount; the JS opens the popup and only submits `order.store` with the `razorpay_payment_id` after a successful payment. - **Graceful cancellation handling** — A cancelled/aborted Razorpay payment still creates a properly recorded `Cancelled` order with its `OrderList` rows and redirects to the failed page — nothing is silently lost, and the cart stays intact for retry. - **Wallet as instant payment** — Wallet checkout verifies the balance

covers `finalAmount`, debits the wallet, and marks the order paid immediately (id `WAL-{rand}`), making wallet the fastest frictionless path. - **Store Pickup treated as paid** — Pickup auto-completes the payment (id `PICKUP-{rand}`, zero shipping), so click-and-collect orders are unambiguous in both client area and admin. - **COD as pending** — COD orders record a `COD-{rand}` id, `is_paid=0`, and `Order received` status, so they appear correctly as payment-pending until admin marks them collected. - **Rewards tied to `isPaid`** — Coupon usage, referral bonuses, and earned DIY points execute exclusively on `isPaid`, guaranteeing no incentive is granted for unpaid or cancelled orders.

STEP 4 Order Success Page (order/success.blade.php)

Route: `GET /order/success/{orderId}` (`order.success`, web.php:686) → `CheckoutController@orderSuccess` → `CheckoutService::getOrderSuccessData()`

Controller Logic Flow: 1. `getOrderSuccessData` loads the order with `orderLists.product.details`, `product.sizeOnlyDetails`, `address.cityRelation`, `address.mandalRelation`, `client`. Recomputes per-item base price (offer>0 ? offer : original; then detail/sizeOnly override), dealer discount when approved, sets `$item->price` post-discount, accumulates `totalClientDiscountApplied`, `subtotalBeforeAllDiscounts`, `totalBasePrice`. 2. The controller additionally computes `extraDiscountAmount` (threshold), `couponDiscountAmount` (= `order->discount_amount`), `earnedDiyPoints` (`WalletTransaction` `diypoint`), `appliedDiyPoints` (`diypoint_usage`).

Plain-English Walkthrough: After a successful order the user lands on the success page. It confirms the order with its id and a full money breakdown recomputed from the database — items, GST, coupon, extra discount, DIY applied, shipping, and paid/due. It also shows how many points the order earned and used. The user can continue shopping or jump straight to the order.

Worked Example (with real numbers): Continuing the ₹670 order from Step 2 (shirt ₹900 incl GST after dealer discount, coupon ₹100, DIY ₹200, shipping ₹70, paid ₹670 via Razorpay): 1. Item line recomputed: base **₹762.71** + GST **₹137.29** = ₹900 incl. 2. Subtotal before all discounts = ₹900. Coupon discount ₹100, DIY applied ₹200, shipping ₹70, paid ₹670, due ₹0. 3. `earnedDiyPoints` read from the `diypoint` wallet transaction = **0** (a coupon was used, so this order wasn't eligible); `appliedDiyPoints` from `diypoint_usage` = **200**. 4. The summary table lists each row above, and the header reads "Your Order Has Been Placed!" with the order id; the user can continue shopping or view the order.

What the page contains: 1. Confirmation header "Your Order Has Been Placed!" 2. Order ID + summary table (items, GST, coupon, extra discount, DIY applied, shipping, paid/due) 3. Continue Shopping / View Order buttons 4. Guest and logged-in variants both supported

Features: - **Confirmation with full money breakdown** — The success page recomputes item prices (offer → variant override → dealer) from the DB and presents the complete summary — subtotal, client discount, coupon, extra discount, DIY applied, shipping, GST, paid/due — so the receipt matches the order record exactly. - **Points visibility right after purchase** — `earnedDiyPoints` and

`appliedDiyPoints` are read from the order's wallet transactions, so the customer immediately sees how many points this order earned or consumed. - **Clear next actions** — Continue Shopping and View Order buttons push the user back into the catalogue or into their order detail, keeping the momentum after a purchase. - **Works for guests too** — The success page renders regardless of session state, so an order completed before logout (or from a shared session) still shows the right confirmation.

STEP 5 Order Failed Page & Payment-failed recovery (order/failed.blade.php)

Routes: `GET /order/failed` (`order.failed`, web.php:684) → `CheckoutController@failed`; `POST /order/payment-failed` (`order.payment-failed`, web.php:713) → `CheckoutController@paymentFailed`

Controller Logic Flow: 1. `failed` reads `order_id` from query or `session('failed_order_id')`; loads the order (if any) with lists + address relations; recomputes `displaySubtotalBeforeClientDiscount`, `extraDiscountAmount`, reads the coupon session, and `appliedDiyPoints`; then forgets `failed_order_id`. No order id → renders with `order = null` (generic error). Logs incoming/session ids for debugging. 2. `paymentFailed` is the AJAX recovery path: `CheckoutService::handlePaymentFailed` — empty cart → error; else sums item totals, creates a Cancelled order (`payment_id='CANCELLED-{rand}'`, `payment_status='Cancelled'`, `is_paid=0`) with its `OrderList` rows, stores `session(['failed_order_id'=>...])`, returns `{success, order_id}`. The page then redirects to `/order/failed`.

Plain-English Walkthrough: If a Razorpay payment is cancelled, a Cancelled order is recorded and the user is sent to the failed page, which shows the would-be order summary so they know what they tried to buy. Their cart is still intact, so they can return to checkout and retry. If no order id exists (for example, the page was opened directly), a generic error panel is shown instead.

What the page contains: 1. Error panel + order summary (same layout as success) 2. Retry-via-checkout / contact-support prompts 3. The cart is preserved on failure paths so the user can retry

Features: - **Both render-time and AJAX failure paths** — A cancelled Razorpay popup (submit without payment id) and an explicit `paymentFailed` AJAX call both create a recorded Cancelled order and funnel to the failed page, so every failure mode is captured consistently. - **Order context preserved** — The failed page loads the cancelled order (when present) with its lists and address relations and re-renders the money breakdown, so users see exactly what would have been charged rather than a blank error. - **Cart preserved for retry** — Failure paths leave the cart intact (only the cancelled order is created), so the user can return to checkout and pay without rebuilding their basket. - **Diagnostic logging** — The controller logs both the incoming and session order ids, aiding support in tracing which attempt failed and why. - **Graceful no-order fallback** — When no order id exists the page renders a generic error panel, so a direct hit on `/order/failed` never crashes.

STEP 6 Track Order (trackorder/track-order.blade.php & track-timeline.blade.php)

Routes: `GET /track-order` (`track.order`, web.php:744) → `ClientOrderController@track`; `GET /track-order/{id}/timeline` (`track.order.timeline`, web.php:745) → `ClientOrderController@timeline`

Controller Logic Flow: 1. `OrderService::getTrackOrders($userId)` loads all orders with `orderItems.product.details.images`, `sizeOnlyDetails`, `color`, `size`, `shippingAddress.mandalRelation/cityRelation`; per item resolves detail/sizeOnly price+offer, clamps max stock, computes `final_price`, `item_total`, `display_image`. 2. `track-order.blade.php` renders each order card: `display_order_id`, status pill, order date, purchased-items preview, tracking number, “View Order” → timeline. Empty state “Your Box is Empty”. 3. `timeline` → `OrderService::getTimelineOrder` (user-scoped `findOrFail`). The view normalizes `order->order_status` via a `match` into canonical steps (Pending/Processing/Shipped/Delivered/Cancelled, with pickup wording), computes `currentStatusIndex`, `progressPercent` (desktop) and `mobileProgressPercent`, then draws a horizontal stepper (active ≤ current, current highlighted) plus readable status descriptions. The tracking ID has a copy-to-clipboard button.

Plain-English Walkthrough: A logged-in user opens Track Order. Every order is a card showing the order id, a status pill, the date, a preview of the purchased items, and a tracking number. Clicking “View Order” opens the timeline: the admin status is normalized into a standard set of steps and a progress bar fills up to the current one. A copy button copies the tracking id to the clipboard. Pickup orders use pickup wording for the steps.

Features: - **Login-gated, user-scoped tracking** — Both routes require the session user and timeline lookups are `findOrFail` scoped to that user, so one customer can never view another’s order by guessing an id. - **Order cards with everything at a glance** — Each card shows the display order id, status pill, date, purchased-item preview with `display_image`, tracking number, and a link into the timeline, so customers see the essentials without opening each order. - **Visual progress stepper** — The timeline normalizes any admin status into canonical steps (Pending → Processing → Shipped → Delivered / Cancelled) and renders a horizontal stepper with the current step highlighted and prior steps filled, making status instantly scannable. - **Desktop and mobile progress variants** — Separate `progressPercent` and `mobileProgressPercent` calculations drive the same stepper at both screen sizes, avoiding broken bars on small viewports. - **Pickup-aware wording** — Pickup orders reword the steps so the terminology matches the store-pickup experience instead of delivery language. - **Copy-to-clipboard tracking ID** — A copy button on the tracking number lets users share it with support or check delivery partners quickly.

PART 5 — CLIENT AREA (ACCOUNT)

STEP 1 Dashboard (client/dashboard.blade.php)

Route: `GET /client/dashboard` (`client.dashboard`, web.php:612) → `DashboardController@index` → `ClientService::getDashboardData()`

Controller Logic Flow: 1. Gate on `session('user_id')`; if the Client is missing, clear session + cookie and redirect home. Merges `getSettingSeoScripts()`. 2. `ClientService::getDashboardData($userId)` aggregates: `totalOrders`, `pendingOrders`, `walletBalance`, `signupBonus`, `totalReferrals` (clients where `referred_by = userId`), `todayOrders` / `weekOrders` / `monthOrders` (date scoped), `totalReferralBonus`, `totalTransactions`, `totalCommission`. 3. Also groups categories into Men/Women sub-main buckets via `getGroupedCategories()` and loads `Popular::all()`.

Plain-English Walkthrough: A logged-in user opens the dashboard. The service aggregates their total/pending orders, wallet balance, referrals, and recent order counts. The page greets them by name and shows animated counter cards (Profile, Orders, Address, DIY Points, Referrals) that link to each area. If the user has no phone number yet, a popup forces them to enter one — validating uniqueness live — because checkout needs it.

What the page contains: 1. Client header (`frontend.client.header`) + mobile-back bar 2. “Welcome Dear, {name}” heading 3. Five counter cards: Profile, Orders, Address, DIY Points (animated count-up to `$walletBalance`), Referrals — each linking to its route 4. A `client_phone` missing alert: a SweetAlert2 modal forces phone entry (with live `check-phone-unique` validation) posting to `client.updatePhone`

Features: - **Rich account statistics at a glance** — The dashboard aggregates total/pending orders, wallet balance, signup bonus, referral counts, today/week/month order counts, referral bonus totals, transaction counts, and commission totals, giving a genuine account overview rather than a static menu. - **Animated counter cards** — Five cards (Profile, Orders, Address, DIY Points, Referrals) count up to their values with animation and link to their respective routes, making the dashboard feel alive and navigable. - **Phone-onboarding enforcement** — When `client_phone` is missing, a SweetAlert2 modal forces entry with live `check-phone-unique` validation and posts to `client.updatePhone` — the enforcement that guarantees checkout’s phone requirement is always satisfied. - **Stale-session self-healing** — If the session user no longer exists the dashboard clears the session and cookie and bounces home, so deleted accounts never linger in the client area. - **Men/Women category grouping** — `getGroupedCategories` builds Men/Women sub-main buckets for the shared layout, keeping the account area’s navigation consistent with the storefront structure.

STEP 2 Profile & Phone Update (client/profile.blade.php)

Routes: `GET /client/profile` (`client.profile` , web.php:613); `POST /client/update-profile` (web.php:617); `POST /client/update-phone` (web.php:618); `POST /check-phone-unique` / `POST /check-email-unique` (web.php:619-620)

Controller Logic Flow: 1. `profile` loads the client + populars and renders the profile page (client card + edit form: name, phone, secondary mobile, avatar image, remove-image toggle). 2. `updateProfile` validates name (letters-only), phone + secondary mobile (Indian regex); `ClientService::updateProfile` updates name/phones, optionally deletes the old image when `remove_client_img=1` , uploads/compresses a new image to R2 (`compressAndUploadToR2('clients')`); refreshes `session('user_name')` . AJAX and normal responses both supported. 3. `updatePhone` validates uniqueness (ignore self) and updates the phone. `checkPhoneUnique` / `checkEmailUnique` power the debounced live checks on login/register forms.

Plain-English Walkthrough: The user opens Profile and sees their info card plus an edit form for name, primary and secondary phone, and avatar. Saving validates the name and Indian phone formats, uploads/compresses a new avatar to R2 (or deletes the old one if asked), and refreshes the name shown in the header. The same uniqueness endpoints power the live “already taken” checks on the register forms.

Features: - **Complete profile editing** — Name, primary phone, secondary mobile, and avatar are all editable from one page, with the account header showing the client card alongside the form. - **Avatar upload to R2** — New avatars are compressed and uploaded to R2 (`compressAndUploadToR2('clients')`) and the old image is deleted when `remove_client_img=1` , so storage stays tidy and images are served from the CDN. - **Live uniqueness validation** — `checkPhoneUnique` / `checkEmailUnique` power debounced checks on login/register forms, catching duplicates the moment they’re typed instead of after submit. - **Session name refresh** — After a profile update `session('user_name')` is rewritten, so the header greeting and account dropdown reflect the new name immediately. - **Dual response support** — Profile updates return either a normal redirect or AJAX JSON, so both the dashboard phone modal and the full profile form use the same controller cleanly.

STEP 3 Orders List & Order Detail (client/orders.blade.php & order-detail.blade.php)

Routes: `GET /client/orders` (`client.orders` , web.php:623) → `ClientOrderController@index` (DataTables); `GET /client/orders/{id}/details` (`client.order.details` , web.php:624) → `show`

Controller Logic Flow: 1. `index` requires login; AJAX branch uses Yajra DataTables over `OrderService::getClientOrdersQuery` (orders desc with `orderLists.product` , `client` , `parentOrder` , `replacements_count`). Columns: index, `order_code` (with Replacement note when

`is_replacement` + `parentOrder`, or "(Original)" when it has replacements), client name, product count, order date, payment status, total (special replacement logic: subtotal + shipping - discount, or `0.00` when "replacement requested"), order status (colored badge from `getStatusMetadata()`), pickup OTP, actions (View / Invoice). Filter columns for order_code, client_name, order_status. 2. Non-AJAX returns the empty `orders` view (DataTables JS populates it). 3. `show` → `OrderService::getOrderDetail`: order with lists.product, address/billing city+mandal relations, client; builds a `$history` array of status milestones (Pending/Processing/Shipping/Delivered/Cancelled with `created_at`, `processing_at`, `shipping_at`, `delivered_at`, `cancelled_at`). 4. `order-detail.blade.php` renders `display_order_id`, billing/shipping details, order items table (SKU, size, price, GST), order instructions, totals (coupon, client discount, extra discount, DIY points, shipping, paid/due), and the status history.

Plain-English Walkthrough: The user opens My Orders. A searchable, sortable table lists every order with its code, date, payment status, total, a colored status badge, the pickup OTP, and View / Invoice buttons. Clicking View opens the detail page: billing and shipping addresses, every item line (SKU, size, price, GST), special instructions, the full totals block, and a history timeline showing when the order was placed, processed, shipped, delivered, or cancelled.

Features: - **Searchable, sortable order list** — Yajra DataTables renders the orders feed with client-side search/sort on order code, client name, and order status, so a customer with many orders can find one fast. - **Replacement-aware presentation** — The `order_code` column annotates replacement orders with their parent and marks originals "(Original)", while totals use special replacement logic (subtotal + shipping − discount, or `0.00` for "replacement requested") so the list always tells the truth about replacement flow. - **Status badges with metadata** — Each order status renders as a colored badge from `getStatusMetadata()`, giving immediate visual feedback on pending/processing/shipped/delivered/cancelled states. - **Pickup OTP on the list** — The pickup OTP column exposes the collection code directly in the list, saving customers a click when picking up in store. - **Full status history timeline** — Order detail builds a milestone array from `created_at`, `processing_at`, `shipping_at`, `delivered_at`, `cancelled_at`, letting customers see when each stage happened. - **Complete order breakdown** — The detail page shows billing/shipping, item lines (SKU, size, price, GST), instructions, and the full totals block (coupon, client discount, extra discount, DIY points, shipping, paid/due) mirroring the invoice.

STEP 4 Shipping / Billing Addresses

Routes: `GET /shipping-address/index` (`home.shippingAddress`, web.php:583) and `/billing-address/index` (`home.billingAddress`, web.php:592) → `AddressController@index`; create/edit/update/delete per address (web.php:585-596); billing variants `billingStore` / `billingUpdate` merge `address_type='billing'` then delegate to the shipping store/update

Controller Logic Flow — `AddressService`: 1. `getShippingAddresses` lists with state/city/mandal relations, latest first. `getBillingAddressesQuery` selects only billing rows. 2. `createAddress` writes fullname, phone, shipping_type, address_type, state_id, city, mandal, mandal_clone, village, pin_code,

flat_no, location_name, landmark; if `syncPhone` and the client has no phone, copies the address phone into `Client.client_phone` (this is why entering an address fixes the checkout phone requirement). 3. `updateAddress` preserves optional fields unless explicitly sent; ownership enforced in the controller (`abort(403)` on mismatch). 4. Dependent dropdowns: `getCities($stateId)`, `getMandals($cityId)`, `getPincodes($cityId)` — exposed via `GET /get-cities/{state}` (web.php:599), `/get-mandals/{city}` (web.php:765/775), `/get-pin-code/{city}` (web.php:601).

Plain-English Walkthrough: The user manages shipping and billing addresses from their account. Each saved address is a card with Edit and Delete. Adding a new one walks through dependent dropdowns — pick a state, then a district, then a mandal — and the PIN code auto-fills. If the user has no phone yet, saving an address can copy the address phone into their profile, which is why adding an address often satisfies the checkout phone requirement.

What the pages contain: 1. Address list (cards) with edit/delete 2. Create/edit forms with State→District→Mandal dependent selects + auto-filled PIN + Indian-PIN validation 3. Billing variants share the same forms

Features: - **Reusable address service** — Shipping and billing flows share one `AddressService`; the billing variants simply merge `address_type='billing'` and delegate, keeping address logic in exactly one place. - **Phone auto-sync from address** — When `syncPhone` is set and the client has no phone, the address phone is copied into `Client.client_phone` — which is exactly why adding an address resolves the checkout phone-required gate. - **Ownership enforced** — Addresses are scoped to the session user and mismatched ids hit `abort(403)`, so users can't read or mutate each other's addresses. - **Dependent state/district/mandal selects** — New/edit forms chain State → District → Mandal via `get-cities` / `get-mandals`, auto-fill the PIN via `get-pin-code`, and validate the Indian PIN format, eliminating typos in the most error-prone fields. - **Field-preserving updates** — `updateAddress` only changes fields explicitly sent, so partial form submissions never wipe optional data like `landmark` or `location_name`.

STEP 5 Wallet (client/wallet.blade.php)

Routes: `GET /wallet` (`wallet.index`, web.php:604); `POST /wallet/add` (`wallet.add`, web.php:605); `POST /wallet/razorpay-success` (`wallet.razorpay.success`, web.php:606); `GET /wallet-transactions/{id}/edit` (web.php:608) → `WalletController` → `WalletService`

Controller Logic Flow: 1. `index` loads client + wallet and renders the wallet page (balance display with count-up, deposit/add funds form). 2. `addAmount` validates `balance >= 1` and `payment_method in [Manual Pay, Razorpay]`; `WalletService::addAmount` creates-or-credits the wallet (`balance += amount`), generates a unique `Rzr-{rand}` payment id, writes a `WalletTransaction` (Completed for Razorpay else Pending), returns the new total. 3. `razorpaySuccess` → `WalletService::processRazorpaySuccess`: in a DB transaction, `firstOrCreate` wallet, increment balance, store payment method/id. Returns success bool for the JS callback. 4. `edit` returns a transaction's JSON only if it belongs to the user (else 403).

Plain-English Walkthrough: The user opens Wallet and sees their balance with an animated count-up. They can add funds with Razorpay (credited immediately) or Manual Pay (shown as Pending until the admin confirms). This same balance is their DIY-points pool, so whatever they top up here becomes spendable points at checkout. A transaction can only be viewed by the user who owns it.

Features: - **Wallet as the universal balance** — The wallet balance is the DIY-points pool used at checkout, so the page's "add funds" flow directly feeds the points/rewards economy. - **Two funding methods** — Razorpay credits the balance immediately (`Completed`); Manual Pay writes a `Pending` transaction the admin confirms later, giving both instant and manual top-up paths. - **Animated balance display** — The balance renders with a count-up animation, matching the dashboard's visual language. - **Secure transaction introspection** — `edit` returns a transaction's JSON only for the owning user (else 403), so the DataTables edit affordance can't leak another user's data. - **Idempotent wallet creation** — Both add paths use `firstOrCreate` for the wallet and `balance += amount`, so the wallet row is created on demand and credits accumulate without overwrite.

STEP 6 Deposits (client/deposit.blade.php)

Route: `GET /deposit` (`deposit.index`, web.php:632) → `DepositController@index`

Controller Logic Flow: AJAX branch returns a DataTables feed of `WalletTransaction` rows where `payment_method in ['Manual Pay','Razorpay']` for the user, rendering a status badge (Completed/Pending). Non-AJAX renders the deposit page (table container + "Add Deposit" button → `wallet.index`). Includes client header, mobile-back, and footer partials.

Plain-English Walkthrough: The user opens Deposits to review their top-up history. Every Manual Pay and Razorpay deposit appears in a searchable table with a Completed/Pending badge, so they can see which top-ups cleared. An "Add Deposit" button jumps them straight to the wallet page to fund it.

Features: - **Deposit history with status badges** — The DataTables feed lists every Manual Pay / Razorpay deposit with a Completed/Pending badge, so users see which top-ups cleared. - **Direct path to funding** — An "Add Deposit" button links straight to `wallet.index`, keeping the top-up journey one click away from the history. - **Full client-area chrome** — The page wraps itself in the client header, mobile-back bar, and footer partials, matching every other account page's layout.

STEP 7 Transactions Ledger (client/transactions.blade.php)

Route: `GET /client/transactions` (`wallet.transaction`, web.php:705) → `TransactionController@index` → `TransactionService`

Controller Logic Flow: AJAX DataTables over `TransactionService::getTransactionsQuery` (id, amount, payment_method, razorpay_payment_id, created_at). `payment_method` is humanized: `diypoint` + `REF-PT-` prefix → "Referral Points"; `diypoint` → "Earned DIY Point";

`diypoint_usage` → “Used DIY Points”; `Signupbonus` → “Signup Points”; `Referralbonus` → “Referral Points”; `ReferralCommission` → “Referral Commission”; else raw. Non-AJAX renders the page; DataTables JS fills the table.

Plain-English Walkthrough: The user opens the transaction ledger to see every movement of their wallet in one searchable table. Internal codes are shown as friendly names — signup bonus, earned DIY points, used DIY points, referral points, referral commission — with amounts and dates, giving a complete audit of what the wallet did and why.

Features: - **One ledger for every wallet movement** — The transaction feed covers signup bonuses, earned DIY points, used DIY points, referral points, and referral commissions, giving a complete audit of the wallet’s lifecycle. - **Humanized payment labels** — Cryptic internal `payment_method` values are mapped to friendly names (“Used DIY Points”, “Referral Commission”, etc.), so the ledger is readable by customers, not just developers. - **Full DataTables capability** — Sorting and searching over amount, method, payment id, and date come for free from the DataTables wrapper.

STEP 8 Commissions (client/commissions.blade.php)

Route: `GET /client/commissions` (`commissions.index`, web.php:730) → `TransactionController@commissions`

Controller Logic Flow: Requires login; AJAX returns a DataTables feed from `TransactionService::getCommissionsQuery`: joins `wallet_transactions → orders → clients → order_lists → products`, filtering `payment_method = 'ReferralCommission'`, `order_id` present, `orders.payment_status = 'Completed'`, with `distinct(orders.order_code)`. Columns: referred client name, commission amount (₹), date (d M Y), order code, order total. Non-AJAX renders the page.

Plain-English Walkthrough: The user opens Commissions to see every referral commission they earned. Each row shows the referred client’s name, the commission amount, the date, the order code, and the order total — and only for orders that were actually paid in full, one row per order, so the list never inflates with pending sales.

Features: - **Commission earnings visible per order** — The join chain walks transactions → orders → clients → order lists → products, so each row shows the referred client, commission amount, date, order code, and order total. - **Paid-orders-only integrity** — The query filters to `ReferralCommission` rows with an order id whose payment is `Completed`, so users only ever see commission on genuinely paid sales. - **Duplicate-safe rows** — `distinct(orders.order_code)` collapses multi-line orders into one commission row, keeping the ledger clean.

STEP 9 Referrals (client/referrals.blade.php)

Routes: `GET /client-referrals` (`client.referral`, web.php:708) → `ReferralController@index`; `GET /user/ref-register` (`referral.register`, web.php:710) → `ReferralController@register`

Controller Logic Flow: 1. `index` requires login; AJAX DataTables feed from `ReferralService::getReferralsQuery`: clients whose `referred_by = userId`, left-joined to `wallet_transactions` (referrer user, referred_client match, payment methods diypoint/ Referralbonus/ReferralCommission), selecting referred name, `points_earned`, `transaction_id`. Displays `transaction_id` or “Pending Order”, and points or “0.00”. 2. `register` (`?reff=CODE`) stores `session(['referred_by' => CODE])` and redirects home with the `reff` param so signup forms pre-fill the code.

Plain-English Walkthrough: The user opens Referrals and sees a “Share Your Link” box. When a friend signs up through that link, the friend appears in the table with the points they earned or “Pending Order” until their purchase rewards the referrer. Visiting the referral link with `?reff=CODE` automatically captures the code and pre-fills the signup form, so the referral is recorded even if the friend registers later.

Features: - **Referral tracking with pending states** — The referral table joins each referred client to their wallet transactions and shows either the reward transaction id or “Pending Order”, making the reward status explicit. - **Share-your-link box** — A prominent “Share Your Link” box plus “Friends Sign Up” and “Earn DIY Points” explainer cards teach and facilitate the referral loop. - **Automatic code capture** — Visiting `/user/ref-register?reff=CODE` stores the code in the session and redirects home with `reff` in the URL, so the signup forms pre-fill the referral field without the user retyping anything. - **Multi-source reward aggregation** — The join covers `diypoint`, `Referralbonus`, and `ReferralCommission` transactions, so the referral list reflects every kind of reward the referrer earned from that signup.

STEP 10 Client Coupons (client/coupon.blade.php & coupon-index.blade.php)

Routes: `GET /user-coupon` (`user.coupon`, web.php:736); `POST /user/coupon/add` (`coupon.add`, web.php:737); `GET /user/coupon/deposit` (`coupon.deposit`, web.php:739)

Controller Logic Flow: 1. `userCoupon` AJAX → DataTables over `CouponService::userCoupon` (`Pin::where(user_id)` with transaction_no, generated_date, coupon_name, coupon_code, amount, coupon_used_date, status) with status badges (Used=success, else warning). Non-AJAX → `CouponService::couponDeposit` (wallet, setting, seo, scripts, grouped sub-main categories, populars, categories) rendering the coupon list + generate form. 2. `addCoupon` validates amount >= 1, then `CouponService::addCoupon` (wallet check + transactional insert of a new `Pin`). `deposit` renders the coupon generation page with the wallet balance shown.

Plain-English Walkthrough: The user opens My Coupons and sees every PIN coupon they have generated, with its code, name, amount, and whether it's used. A generate form lets them enter an amount up to their wallet balance; the wallet is debited and a new coupon code is created instantly. That code can then be applied at checkout like any other coupon, and it stays "Unused" until a paid order consumes it.

Features: - **Generate discount codes from wallet balance** — Users convert wallet funds into transferable PIN coupons (`coupon_code`, `coupon_name`, `amount`), turning their DIY points balance into shareable discounts. - **Full coupon inventory with status** — The DataTables list shows transaction number, generated date, code, name, amount, used date, and a Used/Warning status badge, so users always know which coupons remain spendable. - **Transactional code creation** — `addCoupon` validates the wallet, debits it, and inserts the `Pin` in one transaction (rollback on failure), so a code is never issued without the balance being reserved. - **Balance context on the generation page** — The deposit page renders the current wallet balance beside the generate form, so users know how much they can convert at a glance.

STEP 11 Invoice & PDF Invoice (client/invoice.blade.php & pdf-invoice.blade.php)

Routes: `GET /client/orders/invoice/{id}` (`client.orders.invoice`, web.php:625) → `ClientOrderController@invoice`; `GET /e-invoice/{order_code}` (`client.invoice.view`, web.php:508) → `ClientOrderController@clientInvoiceView`

Controller Logic Flow: 1. `invoice` → `OrderService::getInvoiceData($orderId)`: loads order + client + lists.product.details/sizeOnlyDetails + shippingAddress.city/mandal relations; recomputes base price per item (offer → detail/sizeOnly override → dealer discount when approved), sets `$item->price` post-discount; builds `couponDetails` from `order->discount_amount` + coupon/pin session codes; computes `deliveryDays` from `shippingAddress.cityRelation->delivery_days` and `expectedDate = now()+days` (unless cancelled); recomputes `extraDiscountAmount` and `totalGST`; `isFreeShipping` from the `FreeShipping` threshold on `total_amount`. 2. `clientInvoiceView($order_code)` → `OrderService::getOrderByCode`: order by `order_code` with client, lists.product.details/sizeOnlyDetails, shipping+billing state/city/mandal relations; reads `diyPointsRedeemed` from the `diypoint_usage` transaction; renders the PDF-oriented `pdf-invoice`.

Plain-English Walkthrough: From the order list the user clicks Invoice (or opens the e-invoice link). The page recomputes every line price from the database, shows billing/shipping (or a pickup note), an item table with SKU/HSN/size/price/GST, the full totals block including DIY points redeemed, and a delivery ETA. The layout is print/save-as-PDF friendly, so it doubles as the customer's official record.

Worked Example (with real numbers): Reusing the ₹670 order: the invoice recomputes the shirt at **₹900** incl GST (base ₹762.71 + GST ₹137.29), lists the coupon **₹100**, DIY points redeemed **₹200**, shipping **₹70**, and shows **Paid ₹670 / Due ₹0** . The delivery note uses the shipping city's

`delivery_days` (say 5) to show `expectedDate = today + 5 days`. HSN appears only because this item has one. Printed/saved, this becomes the customer's official record.

What the pages contain: 1. Invoice logo + billing/shipping details (or pickup note) 2. Order-details table (SKU, HSN only when any item has one, size, price, GST) 3. Totals block (client discount, coupon, extra discount, DIY points, shipping, paid/due) 4. Delivery-days note + order instructions 5. Print/save-as-PDF friendly layout

Features: - **Two invoice renderings, one data source** — The printable invoice (by order id) and the PDF-oriented invoice (by order code, `/e-invoice/{order_code}`) both derive from `OrderService`, so figures are consistent whichever URL is used. - **Accurate per-item pricing** — Base prices are recomputed with the same cascade (offer → detail/sizeOnly override → dealer discount) used at checkout, so the invoice's line totals match the paid amount. - **Delivery ETA on the invoice** — `deliveryDays` from the shipping city relation drives an `expectedDate` note (hidden for cancelled orders), giving customers a concrete arrival estimate on the document. - **Coupon & DIY transparency** — `couponDetails` and `diyPointsRedeemed` are both surfaced, so every deduction on the invoice is explained. - **Clean, print-friendly layout** — HSN numbers appear only when any item has one, and the whole document is styled for print/save-as-PDF, so customers and accountants get a clean record.

PART 6 — CONTENT & STATIC PAGES

STEP 1 About / Company Overview (pages/about.blade.php & overview.blade.php)

Routes: `GET /about` (`about`, web.php:500) → `PageController@about`; `GET /company-overview` (`overview`, web.php:501) → `PageController@function`

Controller Logic Flow: `ContentService::getAbout()` returns `About::latest()->first()` + populars + categories; the about view renders the hero and about title/image sections (`title`, `title_1..title_3`). `ContentService::getCompanyOverview()` returns `Service::latest()->first()` (the overview uses a Service record as its content source) with the same sidebar data; the overview view renders `title`, `title_1..title_4` sections.

Plain-English Walkthrough: A visitor clicks About or Company Overview. The controller reads the latest `About` (or `Service`) record and renders its hero and title/image sections with the normal header and footer. The content itself is edited in the admin, so marketing can rewrite the whole company story and it appears on the site immediately with no developer involvement.

How to use: Content is edited in admin (About / Service records). Pages are pure render + shared data.

Features: - **Admin-managed company copy** — The About page renders the latest `About` record (hero + `title_1..title_3` sections) and the Overview page renders a `Service` record (`title_1..title_4`), so both pages are pure content renderers with no per-page code. - **Shared sidebar data** — Both pages receive `populars` + `categories`, keeping the static pages consistent with the storefront navigation without extra queries. - **No business logic to maintain** — Because these are render-only pages, marketing can update the entire company story from the admin and changes go live instantly.

STEP 2 Terms / Privacy / FAQ

Routes: `GET /terms` (`home.terms`, web.php:771) → `PageController@terms`; `GET /privacy` (`home.privacy`, web.php:769) → `PageController@privacy`; `GET /diy/faq` (`diy.faq`, web.php:784) → `PageController@faq`

Controller Logic Flow: 1. `getTerms` / `getPrivacy` only return categories + populars — the text is static Blade content in `terms.blade.php` / `privacy.blade.php`. 2. `getFaq` returns `FaqCategory::with('questions')->whereHas('questions')` (only categories that have questions), plus categories/populars; the view groups questions by `$category->category` in accordions.

Plain-English Walkthrough: A visitor clicks Terms or Privacy and sees static text rendered with the standard layout — the legal copy lives directly in the Blade files. Clicking FAQ loads the admin-managed question categories (only ones that actually have questions) and shows them as expandable accordions, so a long FAQ stays easy to scan.

Features: - **Static legal text where it belongs** — Terms and Privacy bodies live directly in their Blade files, so editing legal copy is a straightforward template change with no DB dependency. - **Admin-managed FAQ with empty hiding** — FAQ content comes from admin `FaqCategory` / `questions`, and `whereHas('questions')` hides categories that have no questions, so empty accordions never appear. - **Accordion grouping** — Questions render grouped under their category in Bootstrap accordions, keeping a long FAQ scannable. - **Consistent layout data** — Categories and populars are passed to all three pages, so navigation and popular-search links stay available on legal pages.

STEP 3 Blogs (pages/blog-index.blade.php & blog-detail.blade.php)

Routes: `GET /blogs` (`blog`, web.php:502) → `BlogController@index`; `GET /blog/{slug}` (`blog-detail`, web.php:503) → `BlogController@show`

Controller Logic Flow: `ContentService::getBlogs()` returns `ContentPage::latest()->get()`; the index lists posts with `Str::limit($bg->title, 40)`. `getBlogDetail($slug)` tries a slug match first (`Str::slug($title) === $slug`), then a raw title match, else 404; the detail view renders `excerpt` + body.

Plain-English Walkthrough: A visitor clicks Blogs and sees a list of the latest content pages. Clicking a post resolves its slug (or falls back to a raw title match so older links keep working), loads the record, and renders the excerpt and body. Unknown slugs return a proper 404.

Features: - **Simple two-view blog** — `/blogs` lists the latest Content Pages (titles capped at 40 chars) and `/blog/{slug}` renders the excerpt + body, giving a complete content publishing loop from admin to page. - **Lenient slug resolution** — The detail route matches on the slugified title first, then falls back to a raw title match, so both clean URLs and older title-encoded URLs keep working. - **404-safe detail pages** — Unmatched slugs return a proper 404 rather than a blank page.

STEP 4 DIY Shorts (pages/diy-short.blade.php)

Route: `GET /diy/shorts` (`diy.short`, web.php:785) → `PageController@diyShort`

Controller Logic Flow: `ContentService::getDiyShort()` returns all `DiyShort::all()` + categories + populars. The view renders a grid of short-video cards; visitors not logged in see a “Login Required” gate instead of the videos; empty content shows “No DIY Shorts available yet.”

Plain-English Walkthrough: A visitor clicks DIY Shorts. If they aren't logged in, they see a "Login Required" gate instead of the videos. Logged-in users see a grid of short videos that the admin uploads; if none exist yet, a friendly "No DIY Shorts available yet" message is shown instead of a broken grid.

Features: - **Login-gated video content** — Visitors who aren't logged in see a "Login Required" gate instead of the videos, protecting the short-video content and driving account creation. - **Admin-driven shorts grid** — All `DiyShort` records render as a responsive video card grid, added and managed entirely in the admin. - **Graceful empty state** — With no shorts published the page shows "No DIY Shorts available yet" instead of a broken grid.

STEP 5 Services (service/index.blade.php & detail.blade.php)

Routes: `GET /services` (`service.list`, web.php:787) → `ServiceController@list`; `GET /service/{slug}` (`service.detail`, web.php:788) → `ServiceController@detail`

Controller Logic Flow: `ContentService::getServices()` returns all `Service::all()`. The index renders cards (image via `ImageHelper::getPhotoUrl`, title, Read More → `/service/{slug}` via `Str::slug($service->title)`) and a Contact Us button that opens a modal pre-filled with the service name. `getServiceDetail($slug)` matches `Str::slug($title) === $slug` (else 404); the detail view renders the main title/image plus up to four title/image sections (`title_1..title_4`), each with the contact modal.

Plain-English Walkthrough: A visitor clicks Services to see a grid of service cards. "Read More" opens the detail page with the main hero plus up to four admin-defined sections. "Contact Us" opens a modal that is pre-filled with that service's name; submitting it sends a contact enquiry that is automatically labelled with the service it came from.

Features: - **Service cards with smart slugs** — Cards render image, title, and a Read More link built from `Str::slug($title)`, so every service gets a clean, shareable URL automatically. - **Context-aware contact modal** — A Contact Us button opens a modal pre-filled with the service name and posts to `contact.store`, so enquiries are automatically labelled with the service they came from. - **Multi-section service pages** — Detail pages render the main hero plus up to four admin-defined title/image sections (`title_1..title_4`), giving rich, brochure-style service pages with zero code. - **404-safe detail matching** — Slugs that don't match any service return a 404 cleanly.

STEP 6 Career (career/index.blade.php)

Routes: `GET /home-career` (`home.career`, web.php:504) → `CareerController@index`; `POST /career` (`career.store`, web.php:507) → `CareerController@store`

Controller Logic Flow: `CareerService::getCareerPage()` returns categories for the shared layout. `submitApplication($request)` creates a `Career` record: name, email, phone, resume file stored to `uploads/` in the public disk, and the message stored as `about`. The controller redirects back with a success flash.

Plain-English Walkthrough: A visitor opens the career page, fills in name, email, phone, uploads a resume (jpg/jpeg/png/pdf), and types a message. Submitting creates an application record, stores the resume file, and redirects back with a success message. The application then appears in the admin Careers list for review.

Features: - **Resume upload handling** — Applications accept jpg/jpeg/png/pdf resumes stored to `uploads/` on the public disk, and the full application (name/email/phone/message) lands in the admin Careers list. - **Simple, reliable submission** — The store action creates the record and redirects back with a success flash, so candidates get instant confirmation.

STEP 7 Contact & Product Enquiry / Notify-Me / Order-back

Routes: `GET /contact` (`contact`, web.php:505) → `ContactController@index`; `POST /contact` (`contact.store`, web.php:506); `POST /enquiry-save` (`enquiry.store`, web.php:510); `POST /orderproductrecieve/save` (`orderproductrecieve.store`, web.php:511-514)

Controller Logic Flow — `NotificationService`: 1. `contactSave`: sanitizes (strip_tags, email filter, phone digits-only), validates (name letters-only, email rfc+dns, Indian phone regex, subject/message lengths), then creates a `Contact` row (with optional `service_name`). 2. `enquirySave`: requires login (401/AJAX-aware); sanitizes and validates; loads the product (name/category/brand) and writes an `Enquiry` row with `product_id`, code, brand, message. This powers the product-detail **Notify Me** buttons. The `notifiedProductIds` in home/shop/offer make the button show as already requested. 3. `orderProductReceiveSave`: requires login; loads product + client, writes an `OrderProductReceive` row (with client name/email/phone), then emails the admin (`OrderProductRequestClientMail` to `$setting->email` cc `email_1`) via Brevo. This is the “notify me when back / order this product” flow; status != Resolved marks it pending in `hasOrderedBack`.

Plain-English Walkthrough: A visitor fills the contact form (or a service’s Contact Us modal) — the submission is sanitized, validated, and stored as a Contact row for the admin inbox. On an out-of-stock product, clicking Notify Me records an enquiry and the button flips to “requested”. Requesting an order-back also emails the admin immediately, and the product page keeps showing “requested” until the request is resolved.

Features: - **Sanitized, validated contact intake** — Contact submissions are sanitized (`strip_tags`, digits-only phone), validated (letters-only name, rfc+dns email, Indian phone regex, length limits), and stored with an optional `service_name` — keeping the admin inbox free of junk. - **Notify-Me enquiry capture** — `enquirySave` (login-gated) writes a rich `Enquiry` row (product id, code, brand, message), and the shared `notifiedProductIds` set turns the product-detail button into an “already requested” state across home/shop/offer. - **Order-back with admin email alert** —

`orderProductReceiveSave` records the request and emails the admin via Brevo (`OrderProductRequestClientMail`), while an unresolved row keeps `hasOrderedBack` true on the product page to suppress duplicates.

STEP 8 Newsletter

Route: `POST /subscribe` (`newsletter.subscribe`, web.php:790) → `NewsletterController@store`

Controller Logic Flow: sanitizes the email, validates required + email + unique against `newsletter_subscribers`, creates a subscriber (`is_active = true`), redirects back with a success flash. The form lives in the header sidebar and footer; on validation failure Laravel redirects back with errors shown in the layout's `$errors` block.

Plain-English Walkthrough: A visitor types an email into the newsletter box in the sidebar or footer and clicks Subscribe. The email is validated and saved as an active subscriber; duplicate emails are rejected with a friendly message. The admin can then send campaigns to the growing subscriber list.

Features: - **Duplicate-safe subscription** — The unique validation rejects already-subscribed emails with a friendly message, preventing the list from accumulating duplicates. - **Two capture points, one store** — The same form in the header sidebar and footer posts to `newsletter.subscribe`, so every capture path feeds the same subscriber table. - **Admin campaign ready** — Subscribers store as `is_active=true`, ready for the admin Newsletter campaign section to target.

FILE INDEX (All Blade Templates)

Total: **56 .blade.php files** across all frontend directories

Auth (2 files)

File	Purpose
`auth/login.blade.php`	Sliding sign-in / sign-up panel
`auth/login-modal.blade.php`	Login modal for guest gating

Home (1 file)

File	Purpose
`home/index.blade.php`	Main storefront landing page

Layouts (2 files)

File	Purpose
`layouts/header.blade.php`	Header, mega-menu, AJAX add-to-cart, login modal
`layouts/footer.blade.php`	Footer, newsletter, mobile bottom-nav

Products (4 files)

File	Purpose
`products/index.blade.php`	Shop / product listing with filters
`products/card.blade.php`	Grid product card (quick-add, wishlist)
`products/list.blade.php`	Horizontal list card variant
`products/detail.blade.php`	Product detail with variants, tabs, Notify-Me

Categories (2 files)

File	Purpose
`categories/show.blade.php`	Simple category page
`categories/products.blade.php`	AJAX category products with filters

Search (1 file)

File	Purpose
`search/index.blade.php`	Search results with filters

Pages (10 files)

File	Purpose
`pages/about.blade.php`	About page
`pages/overview.blade.php`	Company overview
`pages/terms.blade.php`	Terms & conditions (static)
`pages/privacy.blade.php`	Privacy policy (static)
`pages/faq.blade.php`	FAQ accordions
`pages/diy-short.blade.php`	DIY shorts video grid
`pages/offer.blade.php`	Offer promotions page
`pages/contact.blade.php`	Contact form
`pages/blog-index.blade.php`	Blog listing
`pages/blog-detail.blade.php`	Blog detail

Cart (1 file)

File	Purpose
`cart/index.blade.php`	Cart table + summary + coupon

Checkout (1 file)

File	Purpose
`checkout/index.blade.php`	Checkout: addresses, payment, Razorpay

Order (2 files)

File	Purpose
`order/success.blade.php`	Order placed confirmation
`order/failed.blade.php`	Payment failed / cancelled order

Wishlist (1 file)

File	Purpose
`wishlist/index.blade.php`	Wishlist table + mobile cards

Track Order (2 files)

File	Purpose
`trackorder/track-order.blade.php`	Order list for tracking
`trackorder/track-timeline.blade.php`	Status stepper timeline

Service (2 files)

File	Purpose
`service/index.blade.php`	Services listing
`service/detail.blade.php`	Service detail with contact modal

Career (1 file)

File	Purpose
`career/index.blade.php`	Career application form

Client Area (24 files)

File	Purpose
`client/header.blade.php`	Account navigation header
`client/footer.blade.php`	Account footer
`client/mobile-back.blade.php`	Mobile back bar
`client/dashboard.blade.php`	Account dashboard cards
`client/profile.blade.php`	Profile edit
`client/orders.blade.php`	Orders DataTables list
`client/order-detail.blade.php`	Order detail + status history
`client/invoice.blade.php`	Printable invoice
`client/pdf-invoice.blade.php`	PDF-oriented invoice by order code
`client/shipping-addresses.blade.php`	Shipping address list
`client/shipping-create.blade.php`	New shipping address
`client/shipping-edit.blade.php`	Edit shipping address
`client/billing-addresses.blade.php`	Billing address list
`client/billing-create.blade.php`	New billing address
`client/billing-edit.blade.php`	Edit billing address
`client/wallet.blade.php`	Wallet / add funds
`client/deposit.blade.php`	Deposits DataTables
`client/transactions.blade.php`	Transaction ledger
`client/commissions.blade.php`	Referral commissions
`client/referrals.blade.php`	Referral link + table
`client/coupon.blade.php`	Coupon generation (deposit view)

File	Purpose
`client/coupon-index.blade.php`	My coupons list + generate form
`client/forgot-password.blade.php`	Forgot password form
`client/reset-password.blade.php`	Reset password form

Technologies Used Across All Views

Technology	Usage
Laravel Blade	Templating for all pages (no x-app-layout; pages include header/footer partials)
Bootstrap 5	Grid, forms, tables, modals
jQuery	AJAX filtering, cart/wishlist actions, DataTables wiring
Owl Carousel	Desktop product/banner carousels
Swiper	Mobile product carousels
Yajra DataTables	Orders, deposits, transactions, commissions, referrals, coupons
SweetAlert2	Toasts, confirmations, phone-onboarding modal
Razorpay	Order payments + wallet deposits
Font Awesome	Icon set
CSS Variables	Theming (`--primary` , `--secondary` , `--dark` , `--light`)
R2 Storage	Client image uploads (`compressAndUploadToR2`)
BrevoMailService	Order/notification emails
ImageHelper	`getPhotoUrl()` for all stored images

(End of document — 56 blade files documented across 6 parts, every module has a detailed Features section)