Website Flexi Owner Dashboard & Marketplace

Comprehensive Technical Documentation (Pre■Customization Baseline)

1. Introduction & Overview

This document describes the internal architecture of the “Website Flexi Owner Dashboard & Marketplace” WordPress plugin as delivered before any project■specific customizations. It is intended for developers who will maintain, extend, or debug the plugin and need a clear mental model of how files, functions, roles, AJAX flows, JavaScript modules, and CSS assets are wired together. The plugin provides two primary experiences: • An Owner Dashboard for administrators, managers, and selected dashboard users to manage products, review customer■submitted dresses, view orders and statistics, and configure marketplace settings. • A Marketplace / My Account experience for regular customers who can submit their own dresses for sale (“Sell Your Dress”) and manage them in a dedicated “My Dresses” area using the same SPA manage■products module but in user mode.

2. High■Level Architecture

At a high level, the plugin is structured as follows: • Root PHP entry points – Shortcodes, owner dashboard layout, public helper pages (orders, stats, vendor products, email box). • Admin area – A dedicated “System Settings” page that controls marketplace behaviour and roles (managers and dashboard users). • Includes – Shared helpers and settings handlers, including the internal user■type resolution logic and marketplace endpoint setup. • Modules – The manage■products module (PHP + JavaScript) and the shared AJAX layer which powers the SPA table, bulk actions, inline editing, images, and attributes logic. • Assets – Shared JavaScript bootstrappers and theme scripts, plus CSS bundles for owner dashboard, responsive behaviour, and My Account marketplace pages. All frontend product management (for both owners and users) is handled as a small single■page■application (SPA) backed by a central AJAX endpoint file. The owner dashboard simply embeds the appropriate HTML container and includes the JS/CSS that drive the SPA.

3. Roles & Permissions Model

The plugin implements its own logical “user type” layer on top of standard WordPress roles in order to differentiate between: • Manager – either a WordPress administrator or a user explicitly listed in the plugin’s “Managers” setting. • Dashboard User – a non■manager user explicitly allowed to access the Owner Dashboard via the “Dashboard Users” list. • Marketplace User – any other user, typically a WooCommerce customer, who interacts only with the marketplace / My Account views. Central to this is the helper function (located in includes/helpers.php): • wf_od_get_user_type($user_id) – Returns 'manager', 'dashboard', or 'marketplace' based on plugin options and whether the user is an administrator. This user type is consulted throughout the AJAX layer (particularly in modules/shared/ajax/manage-products-ajax.php) to determine whether an operation is allowed, and whether it should be executed in owner mode (full control) or user mode (only on products authored by the current user). Some operations also double■check WordPress capabilities such as current_user_can('edit_products') when appropriate.

4. File■by■File Documentation

4.1 owner-dashboard.php

Purpose: This is the primary frontend entry point for the Owner Dashboard experience. It registers the [owner_dashboard] shortcode, enqueues necessary assets, defines inline AJAX helpers, and renders the tabbed owner dashboard layout in the frontend. Key responsibilities: • Registering the shortcode [owner_dashboard] which outputs the entire owner dashboard interface. • Enqueuing core owner dashboard CSS and JS (owner-style.css, mobile.css, owner-core.js, owner-dashboard-theme.js, and the manage-products JavaScript module when required). • Rendering the main dashboard HTML structure: – A header and navigation area. – Tabs or sections for: • Manage Products (SPA table) • Orders • Stats • Vendor Products (customer dresses under review) • Settings or other internal sections • Defining any inline JavaScript helpers that integrate WordPress' admin-ajax URL and nonce values into the frontend. How it connects to other components: • When the "Manage Products" tab is shown in owner mode, it calls into modules/manage-products/manage-products.php to output the SPA container with data-mode="owner". • It includes or references orders.php, stats.php, and vendor-products.php to render those specific tab contents. • All actions taken in the "Manage Products" tab are forwarded, via JavaScript, to the AJAX layer implemented in modules/shared/ajax/manage-products-ajax.php.

4.2 functions.php

Purpose: This file contains miscellaneous glue code for the plugin, particularly for the user■side "My Dresses" interface, asset loading, WooCommerce integration tweaks, and minor WordPress adjustments. Key responsibilities: • Registering a shortcode such as [styliiiish_user_manage_products] that renders the user■side manage■products interface. Internally this shortcode calls styliiiish_render_manage_products('user') from modules/manage-products/manage-products.php. • Enqueuing frontend assets on specific pages (e.g., a WooCommerce My Account endpoint or a dedicated user dashboard page): – owner-style.css and mobile.css for consistent styling. – myaccount-style.css when on My Account pages. – JavaScript bootstrappers (owner-core.js) and the manage-products.js module. – Third■party libraries such as Select2 and SweetAlert2 where required. • Extending WooCommerce customer capabilities to allow image uploads (e.g. by adding 'upload_files' capability for the customer role when needed for marketplace submissions). • Removing or overriding theme (e.g., Ekart) scripts and styles on the plugin's dashboard / management pages to prevent conflicts or duplicated UI elements. • Adding custom body classes to key pages so that CSS can be targeted accurately. Connections: • Provides the frontend entry point for marketplace users' product management via the shortcode. • Coordinates with includes/marketplace.php to ensure endpoint content uses the correct shortcode. • Ensures that the SPA JavaScript module is only loaded where necessary.

4.3 orders.php

Purpose: Renders a simplified orders dashboard within the Owner Dashboard interface. It allows managers and dashboard users to view a list of WooCommerce orders in a concise table. Key responsibilities: • Providing a function (e.g. styliiiish_render_orders()) that outputs the HTML for the orders section. • Querying WooCommerce orders, typically via wc_get_orders or a WP_Query on the 'shop_order' post type. • Displaying a table of orders with columns such as: – Order number / ID – Customer name or email – Order total – Status – Date – Basic actions (view order details in wp-admin, etc.) • Implementing basic pagination when there are many orders. • Optionally inlining minimal CSS to style this table consistently with owner-style.css. Connections: • Called from owner-dashboard.php when rendering the Orders tab. • Uses WooCommerce APIs rather than direct database queries where possible.

4.4 stats.php

Purpose: Generates high■level statistics for the store and marketplace, presented as summary cards in the Owner Dashboard. Key responsibilities: • Providing a function (e.g. styliiiish_render_stats()) that outputs HTML for statistics cards. • Computing values such as: – Total number of products. – Number of customer■submitted products awaiting review (pending). – Total number of orders. – Sales for the current month. – Total users and/or total WooCommerce customers. • Optionally computing additional marketplace■specific stats (e.g., approved vs. rejected vendor dresses). • Rendering this information in a grid of cards with clear labels and values. Connections: • Called from owner-dashboard.php when rendering the Stats tab. • Relies on WooCommerce and WordPress functions (wp_count_posts, wc_get_orders, get_users, etc.).

4.5 vendor-products.php

Purpose: Manages the review and moderation of customer■submitted products (e.g., dresses added via the marketplace). It gives the owner team a dedicated interface to approve or reject these items. Key responsibilities: • Providing a function (e.g. styliiiish_render_vendor_products()) that outputs cards or a table of pending vendor products. • Fetching products that are in a specific status (such as 'pending') and that represent customer submissions. • Rendering each product with information like: – Product title – Thumbnail image – Price – Product condition or other attributes – Product owner (user who submitted it) – Controls for Approve / Reject • Including inline JavaScript or referencing external JS to call an AJAX action when a product is approved or rejected. • Implementing the AJAX callback (e.g. styliiiish_vendor_moderate_cb) that: – Validates the current user (must be manager / dashboard user). – Updates the product status to 'publish' on approval. – Updates the product status or meta fields on rejection (e.g. saving a reject reason in custom meta). – Optionally logs who approved/rejected using post meta like _styliiiish_approved_by. Connections: • Integrated as a tab or section in the Owner Dashboard layout in owner-dashboard.php. • Uses WooCommerce product helpers and the same product post type as other products.

4.6 email.php

Purpose: Provides a simple email sender box in the Owner Dashboard for internal communication, contact forms, or quick outreach. Key responsibilities: • Defining a function such as styliiiish_render_email_sender() that outputs a small form containing: – Recipient email or subject fields. – Message textarea. – Submit button. • Applying basic styling for the input fields (possibly via inline CSS or leveraging owner-style.css). • Handling form submissions via WordPress (either via admin-post.php or AJAX), performing validation and sending emails using wp_mail(). Connections: • Optional part of the Owner Dashboard; can be included in a dedicated tab or section. • Depends on WordPress' wp_mail configuration for email delivery.

4.7 admin/system-settings.php

Purpose: Implements the backend “System Settings” page for the plugin. It is accessible to administrators through the WordPress admin interface and controls the marketplace mode and role assignments. Key responsibilities: • Registering a submenu page under Plugins or another admin menu item with a slug like websiteflexi-system-settings. • Rendering the settings page by including admin/system-settings-view.php. • Handling form submissions for different tabs: – Enabling or disabling the marketplace globally. – Selecting the “Add Product Mode” (e.g. 'ajax' vs 'old' form■based mode). – Managing the list of Managers (user IDs allowed to act as owners). – Managing the list of Dashboard Users (non■admins allowed to access the Owner Dashboard). • Validating nonces and user capabilities before saving. • Writing settings to the WordPress options table using update_option(). • Redirecting back to the settings page with status messages after save. Connections: • Uses helper functions from includes/settings-handler.php to read/write option values. • The values saved here directly affect behaviour across the plugin (user type resolution, marketplace enablement, and manage■products behaviour).

4.8 admin/system-settings-view.php

Purpose: Contains the HTML and basic PHP for rendering the System Settings UI. It is purely a view layer; all processing logic resides in system-settings.php and settings-handler.php. Key responsibilities: • Rendering tab navigation (Marketplace, Add Product Mode, Managers, Dashboard Access). • Inside each tab, outputting forms and fields such as: – Checkboxes or toggles for enabling the marketplace. – Radio buttons or select fields for choosing add■product mode. – Multi■select or user pickers for Managers and Dashboard Users. • Including nonce fields and ensuring each form submits to system-settings.php for processing. • Displaying status messages passed via query parameters (e.g. wf_msg=success). Connections: • Strictly responsible for HTML output; uses data from settings-handler.php (e.g., current lists of managers/dashboard users).

4.9 includes/settings-handler.php

Purpose: Centralizes interaction with WordPress options for this plugin. It exposes a small API to read and write plugin■specific settings that control marketplace behaviour and role mapping. Key responsibilities: • Defining constants or functions that encapsulate option keys, for example: – wf_od_option_key_marketplace_enabled() → 'sty_mp_enable_marketplace' – wf_od_option_key_add_product_mode() → 'styliiiish_add_product_mode' – wf_od_option_key_manager_ids() → 'styliiiish_manager_ids' – wf_od_option_key_dashboard_ids() → 'styliiiish_dashboard_ids' • Providing getters such as: – websiteflexi_is_marketplace_enabled() : bool – wf_od_get_add_product_mode() : string – wf_od_get_manager_ids() : array of user IDs – wf_od_get_dashboard_ids() : array of user IDs • Normalizing stored options (e.g. ensuring ids are arrays, converting values to boolean or strings as needed). Connections: • Used by system-settings.php to save and read configuration. • Used by helpers.php and the AJAX layer to determine user type and runtime behaviour.

4.10 includes/helpers.php

Purpose: Provides generic helper functions needed across multiple files, with a focus on user lookup and user type resolution. Key responsibilities: • Implementing a safe wrapper for get_option (e.g. wf_od_get_option($key, $default)). • Providing wf_od_get_users_from_ids($ids) which returns structured data for each user ID (name, email, etc.). • Implementing wf_od_get_user_type($user_id) which returns: – 'manager' if the user is in the managers list or is an administrator. – 'dashboard' if the user is in the dashboard users list. – 'marketplace' otherwise. • Potentially offering small formatting helpers or shared logic used in both admin and frontend parts of the plugin. Connections: • Called from manage-products-ajax.php to enforce permissions on AJAX requests. • Used by admin views to build lists of managers and dashboard users with nicer labels.

4.11 includes/marketplace.php

Purpose: Integrates the plugin with WooCommerce My Account by adding a “Sell Your Dress” endpoint and menu item for regular users. Key responsibilities: • Registering a new endpoint via add_rewrite_endpoint('sell-your-dress', EP_ROOT | EP_PAGES). • Filtering WooCommerce account menu items to inject a new menu entry labelled “Sell Your Dress” or similar. • Registering a callback for the endpoint display, e.g. woocommerce_account_sell-your-dress_endpoint, which: – Checks whether the marketplace is enabled via websiteflexi_is_marketplace_enabled(). – If disabled, outputs a friendly message indicating that submissions are temporarily closed. – If enabled, renders the marketplace submission and management experience, often by echoing a shortcode such as [sell_your_dress] or [styliiiish_user_manage_products]. Connections: • Coordinates with functions.php to ensure that the appropriate shortcode renders the user■side manage■products SPA. • Dependent on WooCommerce account endpoint mechanics (permalinks and rewrite rules).

4.12 modules/manage-products/manage-products.php

Purpose: Builds the HTML shell for the manage■products interface used both by owners and marketplace users. The heavy lifting is performed by JavaScript (manage-products.js) and the AJAX backend (manage-products-ajax.php), but this file provides the DOM structure they operate on. Key responsibilities: • Defining styliiiish_render_manage_products($mode) where $mode is either 'owner' or 'user'. • Outputting a wrapper element, for example: • Rendering the static toolbar and controls around the table, such as: – Search input (id="styliiiish-search"). – Category filter select (id="styliiiish-filter-cat"). – Status filter buttons (e.g. Active, Pending, Draft). – Bulk actions dropdown and Apply button. – “Add Product” button that opens a modal or starts a new product creation flow. • Including markup for auxiliary UI components: – Image viewer modal (id="styliiiishImageModal") with a list container (id="styliiiish-images-list"). – Any hidden template rows or placeholders used by JavaScript. Connections: • Called from owner-dashboard.php when rendering the owner manage■products tab ($mode='owner'). • Called from functions.php / WooCommerce endpoint when rendering the user’s “My Dresses” area ($mode='user'). • The JavaScript SPA reads data-mode from the root element to know whether to apply owner or user behaviour.

4.13 modules/manage-products/manage-products-user.php

Purpose: A legacy or alternative implementation of the user■side manage■products interface. Instead of using the unified SPA container, it outputs a specific layout for users that may have existed before the current SPA refactor. Key responsibilities: • Defining a function such as styliiiish_render_user_products() that: – Outputs a toolbar for user filtering and searching. – Renders a container like where products are listed. • Calling a helper such as styliiiish_user_products_content($page, $user_id) to build the table rows or cards for each of the current user’s products. Connections: • In modern usage, the main entry point for user■side product management is via styliiiish_render_manage_products('user'), which uses the shared SPA. This file exists primarily for backward compatibility or reference, and is not the primary flow once the unified SPA is adopted.

4.14 modules/manage-products/manage-products.js

Purpose: Implements the frontend SPA logic for managing products, both in owner mode and user mode. It handles loading product data, applying filters, performing inline edits, managing images, and executing bulk actions – all via AJAX. Key responsibilities: • Exposing a global object such as window.ManageProductsModule with methods including: – init() – cacheDom() – bindEvents() – loadPage(page) – showSkeleton() – applyFilters() – handleBulkAction() – openImageModal(productId) – uploadImage() – setFeaturedImage() – removeImage() – quickUpdateField(productId, field, value) • In init(): – Detecting the root container ('.styliiiish-manage-products-content'). – Reading data-mode="owner|user". – Calling cacheDom() to store references to key elements (search input, filters, buttons). – Calling bindEvents() to attach event listeners. – Triggering initial loadPage(1). • Event handling includes: – Keyup on search with debounce to avoid excessive AJAX calls. – Change on category filter select. – Click on status filter buttons (e.g. Active, Pending, Deactivated). – Click on pagination links to load different pages. – Click on “Add Product” to call the AJAX action that creates a new product and returns its row. – Click on bulk action apply to send selected product IDs to the AJAX bulk handler. – Clicks for opening the image modal, uploading images, setting featured image, and removing images. – Inline changes in simple fields (price, title, etc.) that are saved via quick update AJAX calls. Connections: • Communicates with modules/shared/ajax/manage-products-ajax.php via admin-ajax.php using the actions defined there. • Bootstrapped by assets/js/owner-core.js once the DOM is ready. • Uses SweetAlert2, Select2, and other libraries if enqueued by functions.php or owner-dashboard.php.

4.15 modules/shared/ajax/manage-products-ajax.php

Purpose: This is the central backend controller for all manage■products SPA operations. It exposes a collection of wp_ajax_* handlers that together implement product listing, creation, updating, status changes, deletion, duplication, image handling, attribute handling, and user■side activation/deactivation. Key AJAX actions (examples): • styliiiish_manage_products_list – Inputs: page, search term, category filter, status filter, mode ('owner' or 'user'). – Behaviour: Queries products based on filters and mode, builds an HTML table or card list, and returns it to the SPA. – Includes stats for counts per status so that filter badges can be updated. • styliiiish_add_new_product – Inputs: mode, optional initial data. – Behaviour: Creates a new WooCommerce product post. – If mode='owner', assigns post_author based on the current manager/dashboard user. – If mode='user', ensures post_author is the current marketplace user. – Returns HTML for the new product row so it can be inserted into the table without a full reload. • styliiiish_update_status – Inputs: product_id, new_status (publish, pending, draft, or a custom “deactivated” state), mode. – Behaviour: Validates permissions, then updates post_status and/or custom meta (e.g. _styliiiish_manual_deactivate) accordingly. • styliiiish_delete_product – Inputs: product_id(s), mode. – Behaviour: Ensures the current user may delete the product (owner or post_author), then deletes or trashes it. • styliiiish_duplicate_product – Inputs: product_id. – Behaviour: Clones the product record (along with key meta and taxonomy terms) and returns the new row for display. • styliiiish_quick_update_product – Inputs: product_id, field identifier, new value. – Behaviour: Updates a single property (e.g. post_title, regular_price, condition) without reloading the whole UI. • styliiiish_get_attributes / styliiiish_save_attributes – Behaviour: Retrieve and store WooCommerce product attributes, working together with helpers-attributes.php. • styliiiish_upload_image_custom / styliiiish_get_images / styliiiish_add_image_to_product / styliiiish_set_featured_image / styliiiish_remove_image – Behaviour: Handle attachment uploads, linking images to products, choosing a featured image, and removing images. • styliiiish_get_cats / styliiiish_save_cats – Behaviour: List available product categories and save selected categories for a given product via wp_set_object_terms. • styliiiish_user_activate_product / styliiiish_user_deactivate_product – Inputs: product_id. – Behaviour: Let marketplace users toggle visibility of their own products, respecting any business rules regarding statuses. Additional logic: • styliiiish_build_manage_products_content($paged, $search, $cat, $status_filter, $mode) – Internal function that returns the full HTML for the products list based on the current filters and mode. • styliiiish_get_products_stats() – Computes counts for active, pending, draft, deactivated, etc., to be sent with the list response. • styliiiish_auto_pending_check($product_id) – Applies business rules that may automatically put a product into 'pending' when certain changes are made (for example, when a previously approved dress is heavily edited). Security and permissions: • Each AJAX handler begins by checking: – is_user_logged_in() – Nonce validation if applicable. – wf_od_get_user_type() to identify 'manager', 'dashboard', or 'marketplace' users. – Product ownership when mode='user' to ensure users touch only their own products. Connections: • This file forms the backend half of the SPA, talking directly to manage-products.js on the frontend.

4.16 modules/shared/helpers-attributes.php

Purpose: Provides helper functions for reading and writing WooCommerce product attributes in formats convenient for the SPA interface. Key responsibilities: • styliiiish_get_attributes_text($product_id) – Builds a concise human■readable string representing the product’s active attributes (e.g. "Size: S, M | Color: Red, Blue"). – Used in the products table to give owners and users a quick overview without opening the full edit screen. • styliiiish_update_wc_attributes_from_string($product_id, $data) – Accepts a serialized or structured representation of attributes coming from the SPA. – Splits the input into attribute taxonomies and values. – Uses wp_set_object_terms() to assign terms (e.g. pa_size, pa_color). – Updates the _product_attributes meta so WooCommerce recognizes the attributes as enabled for the product.

4.17 modules/shared/helpers-images.php

Purpose: Abstracts the generation of HTML fragments for product images and simplifies image■related logic shared across multiple AJAX actions. Key responsibilities: • styliiiish_render_product_images_html($product_id) – Retrieves all image attachments associated with the product. – Determines which image is featured. – Outputs HTML markup for thumbnails, including controls to: • Set featured image. • Remove image from the gallery. • Open in a modal if needed. • Used by the image■related AJAX handlers to return updated HTML after an upload, set■featured, or remove operation. Connections: • Called by manage-products-ajax.php in the responses to image management actions. • The returned HTML is inserted into the image modal by manage-products.js.

4.18 assets/js/owner-core.js

Purpose: A lightweight bootstrapper that initializes the manage■products SPA when the appropriate HTML container is present on the page. Key responsibilities: • Running on DOM ready via jQuery. • Checking if an element with the class .styliiiish-manage-products-content exists. • If present and window.ManageProductsModule is defined, calling ManageProductsModule.init(). Connections: • Loaded on both owner and user manage■products pages by owner-dashboard.php and/or functions.php. • Ensures that the SPA only attempts to initialize where the required DOM structure is available.

4.19 assets/js/owner-dashboard-theme.js

Purpose: Contains theme■level JavaScript for the Owner Dashboard, including dark■mode toggling and global helpers that are not specific to the manage■products SPA. Key responsibilities: • Managing a theme storage key (e.g. 'styliiiish_owner_dashboard_theme') in local storage or similar. • Providing applyTheme(mode) to toggle CSS classes on the (e.g. adding or removing 'dark-mode'). • Wiring dark■mode toggle UI elements (switches or buttons) to change the stored theme and reapply it. • Optionally providing smooth scrolling helpers, tab switching helpers, or other UI enhancements used across dashboard sections (e.g. scrollToManageProducts()). Connections: • Loaded alongside owner-core.js on Owner Dashboard pages. • The CSS in owner-style.css includes rules for body.dark-mode to alter background and text colours.

4.20 assets/css/*.css

assets/css/owner-style.css • Defines the primary visual design for the Owner Dashboard, including: – Layout of header, sidebar (if any), and content area. – Styling for the manage■products table (rows, cells, hover states, inline■edit fields). – Styles for buttons, filters, search bars, and bulk action controls. – Card styles for stats and vendor products sections. – Base styles for modals (including the image modal). assets/css/mobile.css • Adds responsive adjustments for smaller screens: – Converting tables into stacked cards where necessary. – Adjusting font sizes and paddings. – Ensuring action buttons remain accessible on mobile devices. assets/css/myaccount-style.css • Targets WooCommerce My Account pages related to the marketplace: – Styling the “Sell Your Dress” page. – Styling the “My Dresses” (user■side manage■products) area. – Ensuring the SPA controls and tables display cleanly within the WooCommerce account layout.

5. Summary and Extension Guidelines

This documentation establishes a detailed baseline understanding of the Website Flexi Owner Dashboard & Marketplace plugin prior to any custom modifications. The core ideas to keep in mind when extending or debugging the plugin are: • There is a unified SPA manage■products module shared between owners and marketplace users, distinguished by a simple mode flag and enforced by backend permission checks. • The System Settings page (admin/system-settings.php + includes/settings-handler.php) controls global marketplace state and the user type model used across the AJAX layer. • Helper files (helpers.php, helpers-attributes.php, helpers-images.php) centralize repeated logic and should be extended rather than duplicated when new behaviours are added. • All destructive or sensitive actions (delete, duplicate, approve, reject, deactivate) flow through the centralized manage-products-ajax.php controller, making it a key point for auditing and extending business rules. When implementing new features, follow these guidelines: 1) Decide whether the feature belongs in owner mode, user mode, or both, and update the SPA (manage-products.js) only where relevant. 2) Add new AJAX actions to modules/shared/ajax/manage-products-ajax.php, using existing handlers as a template for permission checks and responses. 3) Keep the System Settings page as the single source of truth for configuration that affects permissions or global behaviour. 4) Extend the CSS in owner-style.css and myaccount-style.css instead of inlining new styles whenever possible, to maintain a consistent, maintainable design. This document should be revisited after each significant refactor to keep it aligned with the codebase as it evolves.