Developing a WooCommerce Payment Gateway Plugin for ZERTH Pay

This report provides a comprehensive guide for developing a custom WordPress plugin that functions as a WooCommerce payment gateway, specifically integrating with the ZERTH Pay Merchant API. It details the steps necessary to enable a WooCommerce site to process payments through ZERTH Pay, incorporating essential webhook support for transaction notifications. The explanations are structured to be accessible to individuals with foundational WordPress administration knowledge but limited experience in custom plugin development, external API integration, and webhook handling.

Introduction: Your ZERTH Pay Gateway Journey

This guide is designed to empower a developer to build a fully functional WooCommerce payment gateway plugin for ZERTH Pay. It aims to demystify the process, breaking down complex programming concepts into simple, actionable steps. By the end of this journey, a custom plugin will be understood, enabling a WooCommerce store to seamlessly accept payments via ZERTH Pay, complete with crucial webhook support for real-time transaction updates.

The development process outlined herein will enable the creation of a standalone WordPress plugin that integrates with WooCommerce as a payment gateway. This integration facilitates the processing of payments through the ZERTH Pay Merchant API, encompassing payment initiation, response handling, and the reception of transaction updates via webhooks.

The decision to develop a custom payment gateway plugin, rather than utilizing existing third-party solutions, offers significant advantages. A custom plugin provides unparalleled control and customization over the payment experience, allowing for unique features tailored to ZERTH Pay's specific offerings or the store's particular requirements.[1] This approach ensures seamless integration with WooCommerce's native order management, refund processes, and subscription functionalities.[2] Furthermore, direct processor integrations, as facilitated by a custom solution, can potentially lead to lower transaction fees by eliminating intermediary costs.[2] The ability to accept specific regional payment options offered by ZERTH Pay is another strategic benefit.[2] The effort invested in custom development can thus translate into

tangible business advantages beyond mere technical implementation.

Throughout this report, key concepts will be explored:

- **WordPress Plugins:** These are self-contained units of code designed to extend WordPress functionality without altering core files, ensuring system stability and upgrade compatibility.[3]
- **WooCommerce Payment Gateways:** These are specialized plugins that enable various payment methods on a store's checkout page, acting as the interface between the customer and the payment processor.[5]
- **External APIs (Application Programming Interfaces):** These define the rules and protocols for how different applications communicate. In this context, the plugin will use APIs to send requests to ZERTH Pay's servers to initiate and confirm payments.[7]
- **Webhooks:** These represent an event-driven mechanism where ZERTH Pay will automatically notify the store about transaction updates, crucial for maintaining real-time order statuses.[8]

Chapter 1: Laying the Foundation – WordPress Plugin Basics

Before delving into the specifics of a payment gateway, establishing a solid understanding of the fundamental structure and lifecycle of a WordPress plugin is essential. This chapter guides the developer through setting up the development environment and creating the basic framework for the ZERTH Pay plugin.

Setting up Your Development Environment (Local/Staging Site)

Developing directly on a live production site carries significant risks. A local development environment (e.g., using tools like XAMPP, MAMP, or Local by Flywheel) or a staging site (a duplicate of the live site hosted separately) provides a secure sandbox for coding and testing without impacting the operational live store.[1] This isolated environment is critical for safe experimentation, debugging, and conflict resolution.

The consistent recommendation across development guidelines to utilize local or staging environments underscores the inherent fragility of live WordPress installations and the potential for conflicts.[1] This practice serves as a critical risk mitigation strategy. A live site is a production environment, and direct modifications can lead to downtime, data loss, or security vulnerabilities, negatively affecting user experience and revenue. By segregating development from production, developers can iterate, test, and rectify issues without public exposure. This approach prevents "breaking updates" and ensures system stability.[3] This principle extends beyond plugin

development, highlighting the importance of a structured development lifecycle (development, testing, production) for maintaining system integrity and business continuity in all software deployments.

Creating Your Plugin's Folder and Main File

Every WordPress plugin resides within its own unique folder located inside the wp-content/plugins/ directory.[3] Within this dedicated folder, the main PHP file, which WordPress recognizes as the plugin's entry point, is created. It is considered best practice to name this file identically to its containing folder, for instance, zerth-pay-gateway.php within a zerth-pay-gateway folder.[3] This unique naming convention is crucial for preventing conflicts with other installed plugins.[10]

A recommended folder structure promotes modularity and organization, enhancing maintainability and readability.[4] This structure typically includes:

- zerth-pay-gateway/ (the main plugin folder)
  - zerth-pay-gateway.php (the primary plugin file)
  - includes/ (a directory for core functionality files, such as the payment gateway class)
    - class-wc-gateway-zerth.php (the specific payment gateway class file)
  - assets/ (a directory for static assets like CSS, JavaScript, and images)
    - css/
    - js/
    - images/
  - languages/ (a directory for language files, supporting internationalization)
  - README.txt (an optional file providing plugin description and instructions)

The emphasis on a structured folder system, even when a single PHP file could technically function as a plugin, highlights a critical aspect of scalable development.[4] As plugin complexity increases, a flat file structure becomes unwieldy. Modular organization, with dedicated directories for logic (includes), static assets (assets), and translations (languages), significantly improves maintainability, readability, and overall scalability. This design makes it easier to locate specific code, facilitates collaborative development, and simplifies the debugging process. This approach aligns with fundamental software engineering principles such as separation of concerns and modularity, which are vital for the long-term health of any project and effective team collaboration.

Adding the Essential Plugin Header

The plugin header is a critical block of PHP comments positioned at the very top of

the main plugin file (zerth-pay-gateway.php). This header furnishes WordPress with essential information about the plugin, enabling its recognition and activation from the dashboard.[3] Without this header, WordPress will not identify the plugin.[3]

An example of a well-structured plugin header is as follows:

PHP

```php
<?php
/**
 * Plugin Name: ZERTH Pay WooCommerce Gateway
 * Plugin URI:  https://yourwebsite.com/zerth-pay-gateway
 * Description: A WooCommerce payment gateway for ZERTH Pay.
 * Version:     1.0.0
 * Author:      Your Name
 * Author URI:  https://yourwebsite.com
 * License:     GPL-2.0+
 * License URI: http://www.gnu.org/licenses/gpl-2.0.txt
 * Text Domain: zerth-pay-gateway
 * Domain Path: /languages
 * WC requires at least: 3.0
 * WC tested up to: 8.9
 */

if (! defined( 'ABSPATH' ) ) {
    exit; // Exit if accessed directly
}
```

The ABSPATH check, if (! defined('ABSPATH')) { exit; }, is a standard security measure.[1] This check prevents direct access to plugin files via a URL, which could otherwise expose sensitive code or allow malicious execution. It ensures that the plugin's code is executed only within the legitimate context of a loaded WordPress environment. This small yet crucial piece of code significantly enhances the plugin's security posture, mitigating a common attack vector (direct file execution). It represents a foundational security best practice that contributes to a more secure plugin and, by extension, a more secure WordPress site.

The plugin header functions not merely as metadata but as a formal contract between the plugin and the WordPress system.[3] Its absence prevents WordPress from recognizing the plugin.[3] The specific fields within the header (such as Plugin Name,

Plugin URI, Description, Version, Author) serve not only for display in the administrative panel but also for WordPress's internal management, enabling features like automatic updates [10] and compatibility checks (e.g., WC requires at least). This establishes a structured ecosystem where plugins must adhere to specific conventions to function correctly and safely within the WordPress environment. This adherence to platform-specific conventions and APIs is crucial for successful software integration and effective participation within the broader WordPress ecosystem.

Understanding and Using Activation and Deactivation Hooks for Setup and Cleanup

WordPress provides specialized "hooks" that trigger predefined functions when a plugin is activated or deactivated.[12] These hooks are indispensable for performing one-time setup tasks upon activation and for cleaning up resources when the plugin is deactivated.

The **Activation Hook** (register_activation_hook) is triggered when the plugin is activated. This is the opportune moment to establish initial settings, create custom database tables if necessary, or register custom post types.[12] For the ZERTH Pay plugin, this might involve setting default API key placeholders or establishing initial gateway configurations.

An example of an activation function is provided below:

PHP

```
// In zerth-pay-gateway.php
register_activation_hook( __FILE__, 'zerth_pay_activate' );
function zerth_pay_activate() {
    // Example: Add default options or check for WooCommerce dependency
    if (! class_exists( 'WooCommerce' ) ) {
        deactivate_plugins( plugin_basename( __FILE__ ) );
            wp_die( __( 'ZERTH Pay Gateway requires WooCommerce to be installed and active.', 'zerth-pay-gateway' ) );
    }
    // Add default settings if they don't already exist
    add_option( 'zerth_pay_gateway_settings', array(
```

```
        'enabled' => 'no',
        'title'   => 'ZERTH Pay',
        'description' => 'Pay securely with ZERTH Pay.',
        'api_key' => '',
        'secret_key' => '',
        'test_mode' => 'yes',
    ) );
}
```

The **Deactivation Hook** (register_deactivation_hook) is triggered when the plugin is deactivated. Its purpose is to perform graceful cleanup, such as removing any temporary data or options that are no longer required.[12] This action ensures a "clutter-free WordPress environment".[13] It is important to note that deactivation differs from uninstallation; deactivation typically preserves core data, whereas uninstallation aims to remove all plugin-related data.[12] For this plugin, temporary options might be removed, but critical transaction logs would typically be retained.

An example of a deactivation function is shown below:

PHP

```
// In zerth-pay-gateway.php
register_deactivation_hook( __FILE__, 'zerth_pay_deactivate' );
function zerth_pay_deactivate() {
    // Placeholder for deactivation tasks
    // Example: Clean up temporary options, but generally avoid deleting user-configured settings here.
    // delete_option( 'zerth_pay_gateway_temp_data' );
}
```

The distinction between deactivation and uninstallation hooks highlights a critical consideration for user experience and data integrity.[12] Users frequently deactivate plugins temporarily for troubleshooting or to disable a feature. If deactivation resulted in the loss of all settings, it would lead to a frustrating experience upon reactivation. Uninstallation, conversely, signals a user's intent for complete removal. Proper utilization of these hooks ensures a superior user experience and effective data preservation. This practice fosters trust in the plugin, as users are assured that their configurations will not be lost upon deactivation. Conversely, inadequate cleanup,

such as leaving "orphaned" database entries, can lead to database bloat and potential performance issues over time, even if the plugin remains inactive. This reflects robust software design principles where applications are engineered for clean installation, operation, and removal, minimizing side effects and contributing to overall system stability.

Chapter 2: Building Your WooCommerce Payment Gateway

With the foundational plugin structure established, the focus shifts to transforming it into a functional WooCommerce payment gateway. WooCommerce provides a powerful base class for this purpose, simplifying the development process significantly.

Understanding the WC_Payment_Gateway Class: The Heart of Your Gateway

WooCommerce payment gateways are implemented as PHP classes that **extend** the WC_Payment_Gateway base class.[1] This inheritance is a fundamental aspect of the design, providing the custom gateway with access to all the necessary methods and properties for handling payment logic, settings, and seamless integration with WooCommerce's checkout flow. This approach can be conceptualized as utilizing a blueprint that furnishes all the standard tools required for a payment method.

The custom gateway class typically resides within the includes/ folder, for example, in a file named includes/class-wc-gateway-zerth.php.[9] The requirement to extend WC_Payment_Gateway is a prime illustration of Object-Oriented Programming (OOP) principles applied within the WordPress ecosystem.[1] This architectural decision means that by extending the base class, the custom gateway automatically inherits a substantial amount of pre-built WooCommerce functionality, including its settings API and order handling methods. This significantly reduces the amount of new code that needs to be written and ensures consistency with other WooCommerce gateways. This OOP approach fosters modularity and maintainability, leading to a more stable WooCommerce ecosystem because all payment gateways adhere to a common interface, which minimizes potential conflicts with each other or with future WooCommerce updates. Crucially, this simplifies development, allowing the developer to concentrate on the unique aspects of the ZERTH Pay integration rather than recreating core payment logic.

Initializing Your Gateway: __construct() and Essential Properties

Within the custom gateway class, the __construct() method serves as the initialization point. Here, fundamental information and properties are established that WooCommerce requires to recognize and display the payment method.[6] These

properties include a unique identifier, a display title, and a descriptive summary.

Key properties to define include:

- $this->id: A unique identifier for the gateway, such as 'zerth_pay', used internally by WooCommerce.[5]
- $this->icon: An optional URL pointing to an image icon that will be displayed alongside the gateway's name on the frontend checkout page.[5]
- $this->has_fields: A boolean value. Setting this to true indicates that the gateway requires custom fields to be displayed directly on the checkout page (e.g., for direct credit card input). If ZERTH Pay operates via an offsite redirect or an iframe, this might be false or true depending on the exact integration type.[5]
- $this->method_title: The title of the payment method as it appears to the merchant within the WooCommerce admin settings.[5]
- $this->method_description: A brief description for the payment method displayed in the admin settings.[6]
- $this->supports: An array specifying the features the gateway supports, such as 'products', 'refunds', or 'pre-orders'.

An example of the __construct() method structure (within includes/class-wc-gateway-zerth-pay.php) is provided:

PHP

```php
<?php
if (! defined( 'ABSPATH' ) ) {
    exit; // Exit if accessed directly
}

class WC_Gateway_Zerth_Pay extends WC_Payment_Gateway {

    public function __construct() {
        $this->id                 = 'zerth_pay';
        $this->icon               = plugins_url( 'assets/images/zerth-pay-icon.png', dirname( __FILE__ ) ); // Path to your icon
        $this->has_fields         = false; // Set to true if you need custom fields on checkout
        $this->method_title       = __( 'ZERTH Pay Gateway', 'zerth-pay-gateway' );
        $this->method_description = __( 'Accept payments securely via ZERTH Pay.',
```

```
'zerth-pay-gateway' );

        // Define supported features
        $this->supports = array(
            'products',
            'refunds', // If ZERTH Pay API supports refunds
            'pre-orders', // If applicable
        );

        // Load the settings API
        $this->init_form_fields();
        $this->init_settings();

        // Get settings
        $this->title       = $this->get_option( 'title' );
        $this->description = $this->get_option( 'description' );
        $this->enabled     = $this->get_option( 'enabled' );

        // Hooks
            add_action( 'woocommerce_update_options_payment_gateways_'. $this->id, array( $this,
'process_admin_options' ) );
        // Additional hooks can be added here for frontend display or validation.
    }
}
```

The properties such as method_title, method_description, and title, description define how the payment gateway is presented to both the store administrator and the customer.[5] This separation is crucial for effective user experience and administrative clarity. The merchant requires technical details and configuration options, while the customer needs a clear and concise payment option. This dual interface design pattern is common in user-facing applications, where different user roles necessitate tailored interfaces and information presentation.

Key WC_Payment_Gateway Methods and Their Purpose

Understanding the core methods of the WC_Payment_Gateway class is essential for building a functional payment gateway. The table below provides a concise overview of these methods, their purpose, and when they are typically invoked. This focused presentation helps to clarify the gateway's lifecycle and indicates where custom logic

should be implemented.

Method Name	Purpose	When Called	Example Use Case for ZERTH Pay
__construct()	Initializes the gateway, defines basic properties (ID, title, description), loads settings, and sets up hooks.	When the gateway class is instantiated by WooCommerce.	Setting $this->id, $this->method_title, loading API keys from settings.
init_form_fields()	Defines the settings fields displayed in the WooCommerce admin panel for this gateway.	During the __construct() method, and when the admin settings page is rendered.	Defining fields for ZERTH Pay API Key, Secret Key, Test Mode checkbox.
init_settings()	Loads the saved settings from the database into the gateway's properties.	During the __construct() method, after init_form_fields().	Populating $this->title, $this->description, $this->enabled from saved options.
admin_options()	Outputs the HTML for the gateway's settings screen in the admin.	When the merchant navigates to the gateway's settings page in WooCommerce.	Displays the form fields defined in init_form_fields() for merchant configuration.
payment_fields()	Outputs any custom HTML fields or information on the checkout page for the customer.	On the checkout page, when this payment method is selected.	Displaying a ZERTH Pay logo, a brief description, or an embedded iframe for direct payment.
validate_fields()	Performs custom validation on any fields displayed on the checkout page before payment processing.	When the customer clicks "Place Order" and has_fields is true.	Validating a custom reference number or ensuring required fields are filled if $this->has_fields is true.
process_payment($o	Handles the actual	When the customer	Sending payment

rder_id)	payment processing logic, including API calls to ZERTH Pay, updating order status, and redirection.	clicks "Place Order" on the checkout page.	data to ZERTH Pay API, receiving response, marking order as processing or failed.
process_refund($order_id, $amount = null, $reason = '')	Processes refunds via the payment gateway's API.	When a merchant initiates a refund from the WooCommerce order details page.	Sending a refund request to the ZERTH Pay API for a specific order.

Configuring Admin Settings: init_form_fields() for ZERTH Pay API Credentials

The init_form_fields() method is where the settings fields that will appear in the WooCommerce admin panel, specifically under **WooCommerce > Settings > Payments** for the ZERTH Pay gateway, are defined.[1] Merchants will utilize these fields to input their ZERTH Pay API Key, Secret Key, toggle gateway activation, enable test mode, and configure other relevant options.

An example of the init_form_fields() method is provided below:

PHP

```
// Inside WC_Gateway_Zerth_Pay class
public function init_form_fields() {
    $this->form_fields = array(
        'enabled' => array(
            'title'   => __( 'Enable/Disable', 'zerth-pay-gateway' ),
            'type'    => 'checkbox',
            'label'   => __( 'Enable ZERTH Pay Gateway', 'zerth-pay-gateway' ),
            'default' => 'yes'
        ),
        'title' => array(
            'title'       => __( 'Title', 'zerth-pay-gateway' ),
            'type'        => 'text',
                    'description'  =>  __( 'This controls the title which the user sees during checkout.',
```

```
'zerth-pay-gateway' ),
            'default'     => __( 'ZERTH Pay', 'zerth-pay-gateway' ),
            'desc_tip'    => true,
        ),
        'description' => array(
            'title'       => __( 'Description', 'zerth-pay-gateway' ),
            'type'        => 'textarea',
                'description' => __( 'This controls the description which the user sees during checkout.',
'zerth-pay-gateway' ),
            'default'     => __( 'Pay securely with ZERTH Pay.', 'zerth-pay-gateway' ),
        ),
        'api_key' => array(
            'title'       => __( 'ZERTH Pay API Key', 'zerth-pay-gateway' ),
            'type'        => 'password', // Using password type for sensitive keys
            'description' => __( 'Enter your ZERTH Pay Merchant API Key.', 'zerth-pay-gateway' ),
            'default'     => '',
            'desc_tip'    => true,
        ),
        'secret_key' => array(
            'title'       => __( 'ZERTH Pay Secret Key', 'zerth-pay-gateway' ),
            'type'        => 'password', // Using password type for sensitive keys
            'description' => __( 'Enter your ZERTH Pay Merchant Secret Key.', 'zerth-pay-gateway' ),
            'default'     => '',
            'desc_tip'    => true,
        ),
        'test_mode' => array(
            'title'   => __( 'Test Mode', 'zerth-pay-gateway' ),
            'type'    => 'checkbox',
            'label'   => __( 'Enable Test Mode for ZERTH Pay transactions.', 'zerth-pay-gateway' ),
            'default' => 'yes',
                    'description'  => __( 'If enabled, transactions will be processed in test mode.',
'zerth-pay-gateway' ),
        ),
        'webhook_secret' => array(
            'title'       => __( 'Webhook Secret', 'zerth-pay-gateway' ),
            'type'        => 'password',
                'description' => __( 'Enter the secret key provided by ZERTH Pay for webhook validation.',
'zerth-pay-gateway' ),
            'default'     => '',
```

```
            'desc_tip'    => true,
        ),
    );
}
```

The utilization of type: 'password' for API keys, as suggested by the existence of generate_password_html in the WC_Payment_Gateway class documentation [14], represents a fundamental security measure. While the documentation does not explicitly mandate this type, its presence indicates the correct method for handling sensitive inputs like API keys within the administrative interface. This prevents the keys from being visible in plain text within the browser's source code or on the screen, significantly reducing the risk of accidental exposure to unauthorized individuals who might have temporary access to the admin panel. This practice is a foundational step in robust API key management, which is critical for the overall security of a payment gateway.

Displaying Payment Fields on the Checkout Page: payment_fields()

If the ZERTH Pay integration necessitates customer input directly on the WooCommerce checkout page (e.g., a custom reference number or an embedded iframe), the HTML for these fields is defined within the payment_fields() method.[1] For a typical offsite redirect gateway, this method might simply display a descriptive text or a logo.

An example of the payment_fields() method is shown below:

PHP

```
// Inside WC_Gateway_Zerth_Pay class
public function payment_fields() {
    if ( $this->description ) {
        echo wpautop( wp_kses_post( $this->description ) );
    }
    // If ZERTH Pay requires an iframe or specific fields, add their HTML here.
    // Example: echo '<div id="zerth-pay-iframe-container"></div>';
}
```

Processing the Payment: process_payment() and Interacting with ZERTH Pay

The process_payment() method encapsulates the core logic executed when a customer selects the ZERTH Pay gateway and clicks "Place Order".[1] This method receives the $order_id and is responsible for initiating the transaction with the ZERTH Pay API, managing the order status, and directing the customer to the appropriate next step.

The key steps within this method include:

1. **Retrieving the Order Object:** Obtain the WC_Order object using $order = wc_get_order($order_id);.
2. **Preparing Data Payload:** Assemble the necessary data payload as required by the ZERTH Pay API (e.g., order total, currency, unique order ID, customer details).
3. **Making API Call:** Execute a secure HTTP POST request to the designated ZERTH Pay API endpoint (details covered in Chapter 3).
4. **Handling API Response:**
   - **Success:** If the API call is successful and payment is confirmed, mark the order as processing or completed using $order->payment_complete();.[5] This function automatically handles stock reduction and status transitions.
   - **Pending:** If payment requires manual verification or is in a pending state from ZERTH Pay, set the order status to on-hold using $order->update_status('on-hold');.[5]
   - **Failure:** If the payment fails, display a WooCommerce notice using wc_add_notice() and return a failure array.[5]
5. **Stock Management:** Reduce stock levels using wc_reduce_stock_levels($order_id);.
6. **Cart Management:** Empty the customer's cart using WC()->cart->empty_cart();.
7. **Redirection:** Redirect the customer to the "Thank You" page or to ZERTH Pay's dedicated payment page.

An example of the process_payment() method structure is provided:

PHP

```
// Inside WC_Gateway_Zerth_Pay class
public function process_payment( $order_id ) {
    $order = wc_get_order( $order_id );
```

```
    // 1. Prepare data for ZERTH Pay API
    $payload = $this->prepare_zerth_pay_request_data( $order );

    // 2. Make API call to ZERTH Pay (details in Chapter 3)
    $response_data = $this->call_zerth_pay_api( 'payments/initiate', $payload );

        if ( $response_data && isset( $response_data['status'] ) && $response_data['status'] === 'success' ) {
        // Assuming ZERTH Pay provides a redirect URL for offsite payment
            $redirect_url = isset( $response_data['redirect_url'] )? $response_data['redirect_url'] : $this->get_return_url( $order );

        // Mark order status as 'pending' initially, actual completion via webhook
            $order->update_status( 'pending-payment', __( 'Awaiting ZERTH Pay confirmation via webhook.', 'zerth-pay-gateway' ) );
            $order->add_order_note( sprintf( __( 'ZERTH Pay payment initiated. Transaction ID: %s. Awaiting webhook confirmation.', 'zerth-pay-gateway' ), $response_data['transaction_id'] ) );
        $order->set_transaction_id( $response_data['transaction_id'] );
        $order->save();

        // Reduce stock levels
        wc_reduce_stock_levels( $order_id );
        // Empty cart
        WC()->cart->empty_cart();

        return array(
            'result'   => 'success',
            'redirect' => $redirect_url
        );
    } else {
        // Payment initiation failed, handle errors
            $error_message = isset( $response_data['message'] )? $response_data['message'] : __( 'ZERTH Pay payment initiation failed. Please try again.', 'zerth-pay-gateway' );
        wc_add_notice( $error_message, 'error' ); [5, 6]
        $order->update_status( 'failed', $error_message ); [6]
        return array(
            'result' => 'failure',
        );
```

```
    }
}

// Helper function to prepare data (example)
private function prepare_zerth_pay_request_data( $order ) {
    return array(
        'amount'      => $order->get_total(),
        'currency'    => $order->get_currency(),
        'order_id'    => $order->get_id(),
        'customer_id' => $order->get_customer_id(),
        'return_url'  => $this->get_return_url( $order ),
        'webhook_url' => get_rest_url( null, 'zerthpay/v1/webhook' ), // Your webhook listener URL
        // Add more data as required by ZERTH Pay API
    );
}
```

The distinction between $order->update_status() and $order->payment_complete() is critical for accurate order lifecycle management.[5] Using $order->payment_complete() for successful payments ensures that essential actions, such as inventory reduction, are automatically triggered. Conversely, $order->update_status() is a more general function suitable for setting various order states like 'on-hold' or 'failed', where stock adjustments might not be immediately appropriate or might need to be reverted. Properly managing order statuses ensures accurate inventory tracking and a reliable order fulfillment process. Misusing these functions can lead to discrepancies in stock levels, potential overselling, and manual reconciliation complexities, directly impacting the merchant's business operations and customer satisfaction.

Registering Your Custom Gateway with WooCommerce

After defining the WC_Gateway_Zerth_Pay class, it must be registered with WooCommerce so that it appears as a selectable payment option. This is achieved by adding the class name to the woocommerce_payment_gateways filter.[1] This registration typically occurs in the main plugin file, outside the class definition, after the gateway class file has been included.

An example of the registration process (in zerth-pay-gateway.php) is provided:

PHP

```php
// Ensure WooCommerce is active and our gateway class is loaded
add_action( 'plugins_loaded', 'zerth_pay_gateway_init_class' );

function zerth_pay_gateway_init_class() {
    if (! class_exists( 'WC_Payment_Gateway' ) ) {
        return; // WooCommerce not active
    }
    // Include the gateway class file
    require_once plugin_dir_path( __FILE__ ). 'includes/class-wc-gateway-zerth-pay.php';

    // Add the gateway to WooCommerce
    add_filter( 'woocommerce_payment_gateways', 'add_zerth_pay_gateway_class' );
}

function add_zerth_pay_gateway_class( $methods ) {
    $methods = 'WC_Gateway_Zerth_Pay'; // Add your class name here
    return $methods;
}
```

The utilization of the plugins_loaded action [5] and the woocommerce_payment_gateways filter [1] exemplifies the power of the WordPress hook system for extensibility. The plugins_loaded action ensures that all other plugins, including WooCommerce itself, are fully loaded before the custom gateway class is defined. This prevents errors that might occur if the class attempts to extend WC_Payment_Gateway before the base class is available. The filter, on the other hand, allows for the dynamic "injection" of the custom gateway into WooCommerce's array of recognized payment methods without requiring direct modification of core files. This hook-based approach fosters loose coupling between the plugin and WooCommerce core, making the plugin highly compatible and resilient to future WooCommerce updates. This design leads to a stable and future-proof integration, as changes to WooCommerce's internal structure are less likely to disrupt the gateway's functionality, provided adherence to the documented API.

Chapter 3: Connecting to the ZERTH Pay Merchant API

The payment gateway's primary function is to communicate with the ZERTH Pay Merchant API to initiate transactions and receive responses. This process involves making secure HTTP requests and managing the exchange of data.

Understanding ZERTH Pay API Interaction (General Principles of API Keys, Request/Response)

An API (Application Programming Interface) defines a set of rules that enable two distinct applications to communicate with each other.[7] In the context of ZERTH Pay, the plugin will send requests (e.g., to initiate a payment) and subsequently receive responses (e.g., indicating payment success or failure). Authentication for these interactions typically requires an API Key and a Secret Key.[7] Data exchange is commonly conducted in JSON format.[7]

Key considerations for API interaction include:

- **API Keys:** These are unique identifiers used for authenticating requests.[7] Their security is paramount.
- **HTTPS:** All communication with ZERTH Pay API endpoints must utilize HTTPS to ensure data encryption during transit.[7]
- **Request/Response Format:** The ZERTH Pay API documentation will specify the required data format for requests (e.g., JSON, XML) and the structure of responses. JSON is a widely adopted format.[7]

The emphasis on HTTPS and secure API key management underscores the critical security implications inherent in handling financial data.[7] HTTPS encrypts data, thereby preventing "man-in-the-middle" attacks.[16] API keys serve as credentials, and their compromise could lead to unauthorized access to a merchant's account or sensitive customer data. Neglecting these security measures creates severe vulnerabilities, potentially resulting in data breaches, financial fraud, and legal consequences, such as PCI compliance issues.[6] Implementing these practices fosters trust with customers and ensures adherence to industry standards, which is fundamental for any payment gateway.

Making Secure HTTP POST Requests in WordPress Using wp_remote_post()

WordPress provides its own HTTP API for making external requests, abstracting away the complexities of cURL or other underlying methods. The wp_remote_post() function is the primary tool for sending data to the ZERTH Pay API.[7] This function is designed with security in mind and handles various request parameters, including headers and body data.

An example of a helper method within the WC_Gateway_Zerth_Pay class for making API calls is provided below:

PHP

```php
// Inside WC_Gateway_Zerth_Pay class (or a private helper method)
private function call_zerth_pay_api( $endpoint, $payload ) {
    $api_key    = $this->get_option( 'api_key' );
    $secret_key = $this->get_option( 'secret_key' );
    $test_mode  = $this->get_option( 'test_mode' ) === 'yes';

    // Determine API URL based on test mode
    $api_url = $test_mode? 'https://test-api.zerthpay.com/' : 'https://api.zerthpay.com/';
    $full_url = trailingslashit( $api_url ). $endpoint;

    $headers = array(
        'Content-Type'  => 'application/json',
         'Authorization' => 'Bearer '. $api_key, // Or whatever ZERTH Pay uses (e.g., Basic Auth, custom header)
         // A signature might need to be generated using the secret_key here, depending on ZERTH Pay's requirements.
    );

    $args = array(
        'method'    => 'POST',
        'headers'   => $headers,
        'body'      => wp_json_encode( $payload ), // Encode payload as JSON
        'timeout'   => 45, // Maximum time in seconds to complete the request
        'sslverify' => true, // Ensure SSL certificate is verified
    );

    $response = wp_remote_post( $full_url, $args ); [17, 18]

    if ( is_wp_error( $response ) ) {
        $error_message = $response->get_error_message();
        error_log( sprintf( 'ZERTH Pay API Error: %s', $error_message ) ); // Log for debugging
                wc_add_notice(  sprintf(  __(  'ZERTH  Pay  API  Error:  %s',  'zerth-pay-gateway'  ), $error_message ), 'error' );
        return false;
    }
```

```
    $body = wp_remote_retrieve_body( $response ); [17]
    $data = json_decode( $body, true );

    // Check HTTP status code (e.g., 200 OK, 400 Bad Request, 500 Server Error)
    $http_code = wp_remote_retrieve_response_code( $response );
    if ( $http_code !== 200 ) {
        $error_message = isset( $data['message'] )? $data['message'] : __( 'Unknown ZERTH Pay API error.', 'zerth-pay-gateway' );
                error_log( sprintf( 'ZERTH Pay API responded with status %d: %s', $http_code, $error_message ) ); // Log for debugging
                wc_add_notice( sprintf( __( 'ZERTH Pay API responded with status %d: %s', 'zerth-pay-gateway' ), $http_code, $error_message ), 'error' );
        return false;
    }

    return $data; // Return decoded API response
}
```

The explicit use of wp_json_encode() for the request body and json_decode() for handling responses is crucial for robust API communication.[7] If the API expects JSON, directly passing an array to the body argument in wp_remote_post() will result in application/x-www-form-urlencoded data. Therefore, it is imperative to explicitly encode the array as JSON and set the Content-Type header to application/json. Similarly, API responses must be decoded from JSON. Correctly handling data formats ensures successful API communication and prevents common "bad request" errors. This practice leads to a reliable integration, as the plugin sends and receives data in the format the ZERTH Pay API expects, thereby avoiding frustrating debugging cycles caused by malformed requests or unparseable responses.

Common wp_remote_post Arguments for API Calls

The wp_remote_post function accepts an array of arguments that allow for fine-grained control over the HTTP request. Understanding these arguments is vital for constructing secure and reliable API calls. The table below outlines the most frequently used arguments, their types, descriptions, and typical values relevant to a payment gateway integration.

Argument Name	Type	Description	Example Value

method	string	The HTTP method for the request.	'POST'
headers	array	An associative array of HTTP headers to send with the request.	array('Content-Type' => 'application/json', 'Authorization' => 'Bearer YOUR_API_KEY')
body	string or array	The request body. For JSON APIs, this should be a JSON-encoded string.	wp_json_encode($payload_array)
timeout	int	The maximum number of seconds to wait for the request to complete.	45 (seconds)
sslverify	bool	Whether to verify the SSL certificate of the remote server. Should always be true for production.	true
blocking	bool	Whether the request should be blocking (wait for response) or non-blocking (fire and forget).	true (typically for payment processing)
redirection	int	The maximum number of redirects to follow.	5
httpversion	string	The HTTP version to use (1.0 or 1.1).	'1.1'
user-agent	string	The User-Agent header for the request.	'WordPress/WooCommerce ZERTH Pay Gateway'

A clear table of essential arguments for wp_remote_post ensures that the developer implements secure and performant API calls from the outset. This approach leads to a more robust integration by addressing common pitfalls such as excessively long timeouts or unverified SSL certificates, which could otherwise compromise security or lead to a poor user experience due to slow processing.

Sending Payment Data and API Keys Securely

The secure transmission of API keys and sensitive payment data (such as transaction amount, currency, and customer details) is paramount. All communication with API endpoints must exclusively utilize HTTPS.[7] API keys should consistently be transmitted within request headers (e.g., Authorization: Bearer YOUR_API_KEY) rather than embedded in the request body or URL parameters.[15] Crucially, API keys must never be hardcoded directly into the plugin's source code; instead, they should be stored securely within the plugin's settings and retrieved dynamically using get_option().[15]

The consistent advice to keep API keys out of public repositories and to rotate them periodically underscores the continuous nature of security threats.[7] Storing keys directly within the code (hardcoding) represents a significant vulnerability, as they become exposed if the code is publicly accessible (e.g., via version control systems). Regular rotation of these keys reduces the "window of exposure" should a key become compromised. Proper API key management significantly reduces the attack surface for the payment gateway. This practice leads to compliance with established security best practices and prevents unauthorized access to the ZERTH Pay merchant account, which could otherwise result in fraudulent transactions or data theft. This emphasizes a proactive, rather than merely reactive, security posture.

Handling API Responses: Success, Failure, and Error Messages

Upon making an API call, a response from ZERTH Pay will be received. This response, typically in JSON format, must be parsed to ascertain success indicators, error codes, and associated messages.[7] It is imperative to first check for network or WordPress-level errors using is_wp_error().[17] Subsequently, the HTTP status code of the response should be examined, followed by parsing the API's specific response body to determine the transaction status.

Updating WooCommerce Order Statuses Based on API Responses (payment_complete(), update_status())

Based on the response received from the ZERTH Pay API, the corresponding WooCommerce order status must be updated. For a successful payment,

$order->payment_complete() should be invoked to mark the order as paid, trigger stock reduction, and initiate other relevant WooCommerce actions.[5] If the payment fails, $order->update_status('failed') should be used, accompanied by the display of an error notice via wc_add_notice().[5] In scenarios where the payment is pending (e.g., requiring further action from ZERTH Pay), the order status should be set to 'on-hold'.[5]

Chapter 4: Implementing ZERTH Pay Webhook Support

Webhooks are indispensable for ZERTH Pay as they facilitate real-time updates on transaction events, ensuring that WooCommerce orders accurately reflect payment statuses without the need for continuous polling.

What Are Webhooks and Why They Are Crucial for ZERTH Pay

A webhook is an automated message dispatched from one application (ZERTH Pay) to another (the WordPress site) when a specific event occurs, such as a payment being successfully completed or a refund being issued.[8] Unlike traditional API interactions where the site would repeatedly "ask" ZERTH Pay for updates (a process known as polling), webhooks operate on an "event-driven" model – ZERTH Pay "pushes" the relevant information to the site instantaneously.[8] This real-time notification mechanism is vital for ZERTH Pay to inform the merchant's site about transaction events, ensuring that order statuses are kept current.

Creating a Custom REST API Endpoint in WordPress to Act as Your Webhook Listener

To receive webhooks from ZERTH Pay, the WordPress site requires a dedicated URL, or "endpoint," to which ZERTH Pay can transmit its messages. The most robust method for establishing this is by registering a custom REST API route within WordPress.[8] This approach enables WordPress to specifically handle incoming POST requests directed to this webhook endpoint.

The key steps involved in creating this listener include:

1. **Hooking into rest_api_init:** This action is the designated point for registering custom REST API routes.[20]
2. **Utilizing register_rest_route():** This function defines the endpoint. It requires a namespace (e.g., 'zerthpay/v1') and a route (e.g., 'webhook').
3. **Specifying methods:** The method should be set to POST, as webhooks typically transmit data via POST requests.[8]
4. **Defining a callback function:** This function will execute when ZERTH Pay sends data to the webhook URL. This is where the incoming payload will be processed.

5. **permission_callback:** For webhooks, this can often be set to __return_true if other security measures, such as HMAC signatures, are relied upon for authentication. Alternatively, custom logic can be implemented to verify the source.[20]

An example of the code for registering a webhook route (to be placed in the main plugin file or a dedicated webhook handler file) is provided:

PHP

```
// In zerth-pay-gateway.php or includes/class-zerth-pay-webhook-handler.php
add_action( 'rest_api_init', 'zerth_pay_register_webhook_route' );

function zerth_pay_register_webhook_route() {
    register_rest_route( 'zerthpay/v1', '/webhook', array(
        'methods'             => 'POST', // Webhooks typically send POST requests [8, 22]
        'callback'            => 'zerth_pay_handle_webhook',
        'permission_callback' => '__return_true', // Authenticity will be validated in the callback
    ) );
}

function zerth_pay_handle_webhook( WP_REST_Request $request ) {
    // Placeholder for webhook processing
    // Details will be covered in "Receiving and parsing incoming webhook data" and "Validating webhook authenticity"
    return new WP_REST_Response( array( 'status' => 'success' ), 200 ); // Respond with 200 OK [22]
}
```

Once registered, the webhook URL will typically be https://yourdomain.com/wp-json/zerthpay/v1/webhook. This URL will be provided to ZERTH Pay within their merchant settings.

Registering a REST API endpoint for webhooks offers a superior approach compared to simply using GET parameters for triggering functions.[8] While GET parameters can technically initiate a function [8], they are not designed for receiving complex or sensitive payloads and offer limited security. REST API endpoints, conversely, are specifically built for handling structured data (such as JSON), supporting appropriate

HTTP methods (like POST), and integrating with robust security practices (including permissions and authentication). This approach leads to a more robust, secure, and maintainable webhook listener. It facilitates better handling of JSON payloads, ensures proper HTTP status code responses, and allows for easier integration with WordPress's built-in security features, thereby preventing data integrity issues and enhancing the system's resilience against malformed requests or potential attacks.

Receiving and Parsing Incoming Webhook Data (JSON Payload)

When ZERTH Pay dispatches a webhook, the transaction data, known as the payload, is typically transmitted as JSON within the body of a POST request.[8] Within the zerth_pay_handle_webhook callback function, it is necessary to retrieve and parse this JSON data.

An example of code for receiving and parsing webhook data is provided:

PHP

```
// Inside zerth_pay_handle_webhook function
$body = $request->get_body(); // Retrieve the raw POST body
$data = json_decode( $body, true ); // Decode JSON into an associative array

if ( empty( $data ) || ! is_array( $data ) ) {
    // Log an error for malformed JSON or empty payload
    error_log( 'ZERTH Pay Webhook: Received empty or malformed payload.' );
    return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Invalid payload' ), 400 );
}

// At this point, $data contains the parsed ZERTH Pay transaction information.
// For example: $transaction_id = $data['transaction_id'];
//              $status = $data['status'];
//              $order_id_from_zerthpay = $data['metadata']['woocommerce_order_id']; // Assuming this metadata is sent
```

The explicit validation for empty($data) or !is_array($data) after json_decode() is essential. json_decode() can return null if the input is invalid or empty, and failing to check for this condition can lead to PHP errors or unexpected behavior when attempting to access array keys on a non-array value. Robust input validation at the

initial stage of the webhook handler ensures a more resilient system. It leads to graceful failure for malformed requests, preventing critical errors that could halt all webhook processing, and ensures that only valid data proceeds through the system, a crucial aspect for financial transactions.

Webhook Security Best Practices Checklist

Implementing robust security measures for the webhook listener is critical to ensure that incoming requests are legitimate and that sensitive data remains protected. The following checklist summarizes essential best practices.

Best Practice	Description	How to Implement (for ZERTH Pay Gateway)
Use HTTPS	All webhook URLs must use HTTPS to encrypt data in transit, preventing eavesdropping and man-in-the-middle attacks.	Ensure your WordPress site is configured with an SSL certificate and your webhook URL starts with https://.[8]
HMAC Signatures	Include a cryptographic signature (HMAC) with each webhook request, generated using a shared secret key, to verify authenticity and data integrity.	Configure a "Webhook Secret" in your plugin's settings and in ZERTH Pay's dashboard. In your webhook handler, calculate the HMAC signature of the received payload using your secret and compare it to the signature in the request header. Reject if mismatch.[8]
Constant-Time Comparison	When comparing HMAC signatures, use a constant-time comparison function (e.g., hash_equals() in PHP) to mitigate timing attacks.	Implement hash_equals($calculated_signature, $received_signature) to prevent attackers from inferring information about the secret key based on comparison time.[16]
IP Whitelisting	If ZERTH Pay provides a list of IP addresses from which webhooks originate, configure	Add server-level firewall rules or implement IP checks in your webhook handler to

	your server or plugin to only accept requests from those IPs.	restrict access to ZERTH Pay's known IP ranges.[16]
Idempotency	Design your webhook handler to process each unique event only once, even if duplicate notifications are received.	Use a unique identifier from the webhook payload (e.g., transaction_id) and store it (e.g., in wp_options, wp_transients, or a custom database table) to track processed events. Ignore subsequent requests for the same ID.[22]
Reject Malformed Requests	Implement strict validation for incoming webhook payloads. Reject requests that are unauthenticated, malformed, or missing critical data.	Respond with appropriate HTTP error codes (e.g., 400 Bad Request, 401 Unauthorized) and log the attempts without processing the payload.[16]
Logging (Metadata Only)	Log webhook activity (timestamps, status, source IP) but avoid logging sensitive payload content to prevent data exposure.	Use error_log() or a dedicated logging plugin to record webhook events, excluding personally identifiable information (PII) or credentials.[16]
Regular Secret Rotation	Periodically change the shared secret key used for HMAC signatures to limit the risk of long-term exposure if compromised.	Establish a process to update the webhook secret in both ZERTH Pay's system and your plugin's settings.[15]

The consistent emphasis on multiple security measures, including HTTPS, HMAC signatures, IP whitelisting, and idempotency, highlights that security is a continuous, multi-layered process.[8] No single security measure is entirely foolproof; each practice addresses a specific threat. For instance, HTTPS protects data during transit, HMAC verifies the sender's authenticity and data integrity, IP whitelisting restricts the origin of requests, and idempotency prevents the processing of duplicate notifications. These measures are complementary and collectively form a robust defense. A comprehensive security checklist ensures a proactive approach to protecting financial transactions. This leads to a highly secure payment system that prevents fraud, data

manipulation, and denial-of-service attacks, all of which are critical for maintaining the accuracy of financial operations and customer trust. Neglecting any of these layers increases the attack surface and can lead to potentially catastrophic failures.

Validating Webhook Authenticity: Secret Keys, HMAC Signatures, and HTTPS

It is paramount to verify that an incoming webhook request genuinely originates from ZERTH Pay and has not been subjected to tampering.

1. **HTTPS:** The webhook URL must utilize HTTPS.[8] This encrypts the data during transmission, safeguarding it from interception.
2. **Secret Key/HMAC Signature:** ZERTH Pay will likely provide a "secret key" that is configured in both their system and the plugin's settings. With each webhook, ZERTH Pay generates a cryptographic signature (HMAC) using this secret key and includes it in a request header (e.g., X-ZERTHPAY-Signature). The webhook handler must compute the same signature using the received payload and the stored secret key, then compare it to the incoming signature.[8] A mismatch necessitates the rejection of the request.
3. **IP Whitelisting (Optional but Recommended):** If ZERTH Pay furnishes a list of IP addresses from which their webhooks originate, the server or plugin can be configured to accept requests exclusively from those specified IPs.[16]
4. **Idempotency:** Logic must be implemented to manage duplicate webhook notifications.[22] If ZERTH Pay does not receive a 200 OK response, it may retransmit the webhook. The system should utilize a unique transaction ID from the payload to ensure that each event is processed only once.

An example of code for validating webhook authenticity (within zerth_pay_handle_webhook) is provided:

PHP

```
// After decoding $data
$zerth_pay_signature = $request->get_header( 'X-ZerthPay-Signature' ); // Retrieve signature from header
$secret_key = $this->get_option( 'webhook_secret' ); // Retrieve the stored secret key from plugin settings

if ( empty( $zerth_pay_signature ) ||
| empty( $secret_key ) ) {
```

```
    error_log( 'ZERTH Pay Webhook: Missing signature or secret key.' );
    return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Unauthorized' ), 401 );
}

// Calculate your own HMAC signature
$calculated_signature = hash_hmac( 'sha256', $body, $secret_key ); // Use the raw body, not decoded $data

// Compare signatures using a constant-time comparison to prevent timing attacks [16]
if (! hash_equals( $calculated_signature, $zerth_pay_signature ) ) {
    error_log( 'ZERTH Pay Webhook: Invalid signature.' );
    return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Unauthorized' ), 401 );
}

// Implement idempotency: Check if this transaction_id has already been processed [22]
$transaction_id = isset( $data['transaction_id'] )? sanitize_text_field( $data['transaction_id'] ) : '';
if ( empty( $transaction_id ) ) {
    error_log( 'ZERTH Pay Webhook: Missing transaction ID in payload for idempotency check.' );
    return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Missing Transaction ID' ), 400 );
}

if ( get_transient( 'zerth_pay_webhook_processed_'. $transaction_id ) ) {
    error_log( 'ZERTH Pay Webhook: Duplicate transaction ID received: '. $transaction_id );
    return new WP_REST_Response( array( 'status' => 'success', 'message' => 'Already processed' ), 200 );
}
set_transient( 'zerth_pay_webhook_processed_'. $transaction_id, true, DAY_IN_SECONDS ); // Store for a day to prevent replay attacks
```

The integration of HTTPS, HMAC signatures, and idempotency forms a robust, layered security model.[8] This comprehensive approach ensures protection against various attack vectors, including eavesdropping, spoofing, tampering, and replay attacks, while also preventing data inconsistencies such as duplicate orders or refunds. For a payment gateway, this level of robust security is essential for maintaining financial accuracy and customer trust, directly impacting the business's financial stability and reputation. Neglecting any of these security layers significantly increases the attack surface and can lead to potentially catastrophic failures.

Processing ZERTH Pay Transaction Events and Updating WooCommerce Order

Statuses

Once the webhook is validated, the data contained within the payload is utilized to update the corresponding WooCommerce order. This typically involves extracting the order ID (which should have been passed to ZERTH Pay during payment initiation as metadata) and the transaction status.

The key steps for processing transaction events are:

1. **Retrieve Order Object:** Obtain the WC_Order object using the order ID extracted from the webhook payload.
2. **Update Order Status:** Based on the ZERTH Pay transaction status (e.g., completed, failed, refunded), update the WooCommerce order status. This involves using $order->payment_complete(), $order->update_status(), or $order->add_order_note() as appropriate.[5]
3. **Add Order Notes:** Record the webhook event and ZERTH Pay's status as notes within the WooCommerce order for auditing and administrative purposes.
4. **Handle Refunds:** For refund events, ensure the order is correctly marked as refunded and that stock adjustments are made if necessary.

An example of code for processing ZERTH Pay transaction events (within zerth_pay_handle_webhook after validation) is provided:

PHP

```
// Assuming $data contains the parsed webhook payload
$order_id = isset( $data['metadata']['woocommerce_order_id'] )? intval( $data['metadata']['woocommerce_order_id'] ) : 0;
$zerth_pay_status = isset( $data['status'] )? sanitize_text_field( $data['status'] ) : '';
$zerth_pay_transaction_id = isset( $data['transaction_id'] )? sanitize_text_field( $data['transaction_id'] ) : '';

if (! $order_id ) {
    error_log( 'ZERTH Pay Webhook: Missing WooCommerce Order ID in payload.' );
    return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Missing Order ID' ), 400 );
}
```

```
$order = wc_get_order( $order_id );
if (! $order ) {
    error_log( 'ZERTH Pay Webhook: Order not found for ID '. $order_id );
     return new WP_REST_Response( array( 'status' => 'error', 'message' => 'Order not found' ),
404 );
}

switch ( $zerth_pay_status ) {
    case 'completed':
        if (! $order->is_paid() ) {
            $order->payment_complete( $zerth_pay_transaction_id ); // Mark as paid [5, 6]
                    $order->add_order_note( sprintf( __( 'ZERTH Pay webhook: Payment completed.
Transaction ID: %s', 'zerth-pay-gateway' ), $zerth_pay_transaction_id ) );
        }
        break;
    case 'failed':
                $order->update_status( 'failed', sprintf( __( 'ZERTH Pay webhook: Payment failed.
Transaction ID: %s', 'zerth-pay-gateway' ), $zerth_pay_transaction_id ) );
        break;
    case 'refunded':
            $order->update_status( 'refunded', sprintf( __( 'ZERTH Pay webhook: Payment refunded.
Transaction ID: %s', 'zerth-pay-gateway' ), $zerth_pay_transaction_id ) );
        // Additional logic might be required here for partial refunds or stock management.
        break;
    // Include additional cases for other ZERTH Pay statuses (e.g., 'pending', 'cancelled').
    default:
         $order->add_order_note( sprintf( __( 'ZERTH Pay webhook: Unknown status "%s" received
for transaction ID: %s', 'zerth-pay-gateway' ), $zerth_pay_status, $zerth_pay_transaction_id ) );
        break;
}

return new WP_REST_Response( array( 'status' => 'success' ), 200 ); // Always respond with 200
OK [22]
```

Chapter 5: Essential Best Practices, Security, and Debugging

Developing a functional plugin is merely one aspect; creating a secure, performant, and maintainable solution is equally critical. Adhering to these best practices is fundamental for the long-term success and reliability of the ZERTH Pay gateway.

Plugin Security: Input Sanitization, Output Escaping, HTTPS, and PCI Compliance Considerations

Security is paramount for any payment gateway. A multi-layered approach is essential to protect sensitive data and maintain system integrity.

- **Input Sanitization:** All user input, whether from administrative settings or form fields, must be sanitized before processing or storage in the database.[3] WordPress provides functions such as sanitize_text_field(), sanitize_email(), and absint() for this purpose. This practice prevents malicious data injection, including SQL injection and Cross-Site Scripting (XSS) attacks.[4]
- **Output Escaping:** Any data intended for display on the screen, whether on the frontend or within the administrative interface, must be escaped.[11] Functions like esc_html(), esc_attr(), esc_url(), and wp_kses_post() are used to prevent XSS attacks, where malicious scripts could be injected into the rendered page.
- **Nonces:** WordPress nonces should be utilized for all forms or URLs that trigger actions.[11] Nonces provide a layer of protection against Cross-Site Request Forgery (CSRF) attacks.
- **HTTPS:** The entire WooCommerce site, particularly the checkout process, must employ HTTPS.[6] This encrypts all data transmitted between the user's browser and the server, safeguarding sensitive information like credit card details.
- **PCI Compliance:** For a *direct* payment gateway, where payment fields are displayed on the site, Payment Card Industry Data Security Standard (PCI DSS) compliance may be required.[6] This standard outlines security requirements for handling credit card information. While form-based or iFrame-based gateways typically offload much of this responsibility, direct integrations place it on the merchant. It is advisable to consult ZERTH Pay's documentation and a security expert for specific PCI requirements.
- **API Key Management:** As previously discussed, API keys must be stored securely, never hardcoded, and their rotation should be considered regularly as a best practice.[7]

The consistent emphasis on security measures across multiple guidelines, including sanitization, escaping, HTTPS, PCI considerations, and API key management, indicates that security is not a singular feature but a continuous, multi-layered process.[1] Neglecting one area, such as input sanitization, can undermine efforts in another, like HTTPS. Each measure protects against a different attack vector. This comprehensive, multi-layered security approach results in a significantly more robust and trustworthy payment gateway. It leads to compliance with industry standards, prevents data breaches and financial fraud, and ultimately builds customer confidence. For a

payment gateway, security failures can have catastrophic consequences, including legal liabilities, reputational damage, and severe financial penalties.[6] This underscores that security is an ongoing commitment, not a one-time task.

Code Quality and Performance: Modular Design, Unique Naming, and Efficient Coding

A well-engineered plugin is not merely functional but also comprehensible, maintainable, and efficient in its performance.

- **Modular and Organized Code:** Code should be modular and organized using a clear folder structure, such as includes/ and assets/.[3] Complex logic should be broken down into smaller, reusable functions or methods.
- **WordPress Coding Standards:** Adherence to WordPress coding standards ensures consistency and readability.[4]
- **Unique Naming Conventions:** All custom functions, classes, variables, and CSS selectors should be prefixed with the plugin's unique name (e.g., zerth_pay_).[10] This practice is crucial for preventing conflicts with other plugins or themes within the global WordPress environment.
- **Appropriate Use of WordPress Hooks:** Actions and filters should be leveraged to integrate functionality without directly modifying core WordPress or WooCommerce files.[3] Adopting a "just in time" (JIT) mentality, where code executes only when necessary, optimizes performance.[10]
- **Performance Optimization:** Database queries and code execution should be minimized where feasible.[11] For API calls, consider caching responses if data is not real-time, or implementing rate limiting to prevent overwhelming the API.[2]

The advice regarding modularity, unique naming, and proper hook usage is not merely about "clean code"; it pertains to future-proofing and ecosystem compatibility.[3] These practices collectively reduce the likelihood of conflicts with other plugins or themes and simplify debugging and maintenance. They also enable easier updates for both the plugin itself and the core WordPress/WooCommerce platforms. High-quality code results in a stable, performant, and maintainable plugin, leading to a superior user experience, fewer support requests, and an extended lifespan for the plugin within the dynamic WordPress ecosystem. Conversely, poorly structured code can lead to "technical debt" and introduce unexpected issues, particularly with WordPress updates.

Error Handling and Logging: Using WP_DEBUG, debug.log, and error_log() for Effective Troubleshooting

When operational issues arise, effective debugging tools become indispensable.

- **WP_DEBUG in wp-config.php:** Setting define('WP_DEBUG', true); in the wp-config.php file enables the display of PHP errors, warnings, and notices directly on the site.[24] This setting should always be disabled on live production sites for security and user experience reasons.
- **WP_DEBUG_LOG:** Setting define('WP_DEBUG_LOG', true); directs all errors to be saved in a debug.log file located in wp-content/.[24] This is invaluable for debugging AJAX requests or background processes, such as webhooks, which do not output directly to the browser.
- **error_log():** PHP's error_log() function can be used to strategically log messages, variable values, or execution points to the debug.log file.[24] This assists in tracing the flow of code execution.
- **Browser Developer Tools:** For frontend issues involving JavaScript or CSS, the browser's developer console (console.log()) is used to inspect elements, monitor network requests, and identify JavaScript errors.[24]
- **Debugging Plugins:** Plugins such as Query Monitor offer a more comprehensive debugging interface directly within the WordPress administrative area.[24]

The layered approach to debugging, encompassing WP_DEBUG, error_log(), browser tools, and specialized plugins, reflects the inherent complexity of the WordPress environment.[24] No single tool is sufficient for all debugging scenarios; backend PHP errors require server-side logging, while frontend JavaScript errors necessitate browser tools. Background processes like webhooks specifically benefit from log files. A comprehensive debugging toolkit facilitates faster problem identification and resolution, leading to reduced downtime and a more stable plugin, which is critically important for a payment gateway. Conversely, neglecting robust debugging practices can result in prolonged troubleshooting, user frustration, and potential loss of business if issues persist on a live site.

Thorough Testing: Local, Staging, and Live Environment Considerations

Prior to deploying the plugin to a live environment, rigorous testing is an absolute necessity.

- **Local/Staging Site:** Initial development and testing must always be performed on a local or staging environment.[1]
- **Test Transactions:** Conduct numerous test transactions through the ZERTH Pay gateway, covering successful payments, failures, pending states, and refunds.[1] Verify that order statuses, stock levels, and customer notifications are accurate.
- **Webhook Testing:** Utilize tools such as RequestBin or Webhook.site to capture

and inspect incoming webhook payloads from ZERTH Pay during testing.[22] This ensures that ZERTH Pay is transmitting data correctly and that the webhook listener is successfully receiving and parsing it.
- **Compatibility:** Test the plugin with various WordPress themes and other commonly used plugins (e.g., caching plugins, security plugins) to identify potential conflicts.[10]
- **Environment Consistency:** Conduct testing on environments that closely replicate the live server's PHP version, WordPress version, and overall server configuration.[10]

The consistent emphasis on testing across diverse environments and scenarios underscores that a payment gateway's reliability is paramount.[1] This signifies that testing extends beyond mere functionality verification; it encompasses ensuring robustness, compatibility, and resilience under real-world conditions. Different environments (local vs. staging vs. live) and various scenarios (success, failure, edge cases) can unveil latent bugs. Thorough, multi-stage testing results in a high-quality, reliable payment gateway. This leads to confidence in transactions, prevents unexpected errors on the live site that could result in lost revenue or customer dissatisfaction, and ultimately protects the merchant's business integrity. For a payment gateway, even minor bugs can have significant financial consequences.

Admin Settings Page

While not explicitly requested in the initial query, a robust payment gateway inherently requires an administrative settings page. This page serves as the interface where the merchant configures their ZERTH Pay API keys, enables or disables the gateway, sets test mode, and potentially configures webhook URLs or other gateway-specific options.[2] The init_form_fields() method, previously discussed, defines these fields, implying the existence of such a configuration interface.

The creation of an admin settings page can be achieved using the WordPress Settings API [23], which involves several key steps:

1. **Adding a Menu Item:** A menu item is added to the WordPress dashboard using functions like add_options_page (for a submenu under "Settings") or add_submenu_page (for a submenu under a custom top-level menu).[23]
2. **Defining Page Content:** The content of the settings page is rendered within a form, utilizing settings_fields() to output necessary hidden fields and nonces for security, and do_settings_sections() to display all registered sections and fields.[23]
3. **Registering Settings, Sections, and Fields:** The actual settings, their organization into sections, and the individual input fields are registered using

register_setting(), add_settings_section(), and add_settings_field() respectively.[23]

4. **Implementing Validation and Sanitization:** Callback functions are implemented to validate and sanitize input data before it is saved to the database.[23]

The necessity of an admin settings page, although an implicit requirement, is a critical feature for both usability and maintainability. Without such a page, the merchant would be compelled to hardcode sensitive API keys or directly modify plugin files, which is insecure, impractical, and deviates from WordPress best practices. The init_form_fields() method itself implies the existence of such a configuration interface. Providing a user-friendly admin settings page enables the plugin to be easily configurable and usable by non-developers. This leads to better adoption and reduces the need for manual code edits, which could otherwise introduce errors or security vulnerabilities. This constitutes a crucial element for a complete and professional plugin.

Conclusion: Launching Your ZERTH Pay Gateway

The journey to develop a custom ZERTH Pay payment gateway for WooCommerce, as detailed in this report, encompasses a comprehensive understanding of WordPress and WooCommerce development principles, external API integration, and critical security practices. The successful implementation of these steps culminates in a functional and robust payment solution.

The development process has involved building a standalone WordPress plugin, integrating it seamlessly as a WooCommerce payment gateway, establishing secure communication with the ZERTH Pay API for transaction processing, and implementing a resilient webhook listener for real-time transaction updates.

Before deploying the plugin to a live production environment, a series of final checks are imperative:

- Ensure that WP_DEBUG is set to false in the wp-config.php file on the live site to prevent the display of sensitive error messages.
- Verify that all ZERTH Pay API keys and secret keys are accurately configured within the plugin's administrative settings for the live environment.
- Confirm that the webhook URL is correctly set up in ZERTH Pay's merchant dashboard and points precisely to the live site's designated webhook endpoint.
- Conduct a final series of live test transactions (utilizing ZERTH Pay's live test credentials, if available, or small real transactions) to ensure end-to-end functionality and data flow.

Ongoing maintenance and support are crucial for the long-term reliability and security of the ZERTH Pay gateway:

- Regularly update the plugin, WordPress core, WooCommerce, and the underlying PHP version to benefit from security patches and performance improvements.
- Continuously monitor the site's error logs (debug.log) for any unexpected issues or anomalies.
- Stay informed about any changes or updates to the ZERTH Pay Merchant API or their webhook specifications to ensure continued compatibility.
- Periodically review and rotate API keys as a fundamental security best practice.

Works cited

1. Add Custom Payment Gateway To WooCommerce - IThemeland, accessed June 9, 2025, https://ithemelandco.com/blog/woocommerce-add-custom-payment-gateway/
2. Create WooCommerce Payment Gateway Plugin: Development Tutorial, accessed June 9, 2025, https://www.brihaspatitech.com/blog/create-woocommerce-payment-gateway-plugin-development-tutorial/
3. A Beginners Guide to WordPress Plugin Development - Bluehost, accessed June 9, 2025, https://www.bluehost.com/blog/wordpress-plugin-development/
4. Creating a WordPress Plugin: From Concept to Launch - Codeable, accessed June 9, 2025, https://www.codeable.io/blog/creating-a-wordpress-plugin/
5. WooCommerce Payment Gateway API - Woo Developer Docs, accessed June 9, 2025, https://developer.woocommerce.com/docs/woocommerce-payment-gateway-api-2/
6. WooCommerce Payment Gateway API | WooCommerce developer ..., accessed June 9, 2025, https://developer.woocommerce.com/docs/features/payments/payment-gateway-api/
7. How to integrate an external API with WordPress - Tremhost News, accessed June 9, 2025, https://tremhost.com/blog/how-to-integrate-an-external-api-with-wordpress/
8. How to Use Webhooks in WordPress? - Verpex, accessed June 9, 2025, https://verpex.com/blog/wordpress-hosting/how-to-use-webhooks-in-wordpress
9. Create a WooCommerce Payment Plugin: Step-by-Step Guide - WorldWin Coder, accessed June 9, 2025, https://worldwincoder.com/blog/create-a-woocommerce-payment-plugin-step-by-step-guide/
10. 9 Tips for WordPress Plugin Development - WebFX, accessed June 9, 2025, https://www.webfx.com/blog/web-design/wordpress-plugin-development-tips/

11. WordPress Plugin Development Best Practices for 2025 - ColorWhistle, accessed June 9, 2025, https://colorwhistle.com/wordpress-plugin-development-best-practices/
12. Activation/Deactivation of a Plugin - wp-kama, accessed June 9, 2025, https://wp-kama.com/handbook/plugin/create/huki-aktivatsii-deaktivatsii
13. Managing the Lifecycle: A Guide to Activation and Deactivation in WordPress Plugin Development | Dhanendran's Blog, accessed June 9, 2025, https://dhanendranrajagopal.me/technology/managing-the-lifecycle-a-guide-to-activation-and-deactivation-in-wordpress-plugin-development/
14. WC_Payment_Gateway - WooCommerce Code Reference, accessed June 9, 2025, https://woocommerce.github.io/code-reference/classes/WC-Payment-Gateway.html
15. WooCommerce REST API: Best Practices and Tips - DhiWise, accessed June 9, 2025, https://www.dhiwise.com/post/woocommerce-rest-api-guide
16. Webhook Security Best Practices and Checklist | Secure Your ..., accessed June 9, 2025, https://www.invicti.com/blog/web-security/webhook-security-best-practices/
17. Sening HTTP requests in WordPress using wp_remote_request ..., accessed June 9, 2025, https://pexetothemes.com/wordpress-functions/wp_remote_request/
18. Posting data to a remote server in WordPress with wp_remote_post - Pexeto Themes, accessed June 9, 2025, https://pexetothemes.com/wordpress-functions/wp_remote_post/
19. Custom API for WP Plugin — WordPress.com, accessed June 9, 2025, https://wordpress.com/plugins/custom-api-for-wp
20. Receiving Stripe Webhooks on a wordpress website - Stack Overflow, accessed June 9, 2025, https://stackoverflow.com/questions/40015091/receiving-stripe-webhooks-on-a-wordpress-website
21. WordPress REST API – custom routes & endpoints, accessed June 9, 2025, https://learn.wordpress.org/tutorial/wordpress-rest-api-custom-routes-endpoints/
22. How to Use WooCommerce Webhooks for Custom Development ..., accessed June 9, 2025, https://pantheon.io/learning-center/wordpress/woocommerce-webhooks
23. 5 Ways to Create a WordPress Plugin Settings Page - Delicious Brains, accessed June 9, 2025, https://deliciousbrains.com/create-wordpress-plugin-settings-page/
24. WordPress Debug Guide: Solve Errors Quickly & Efficiently, accessed June 9, 2025, https://wpwebinfotech.com/blog/wordpress-debug/
25. debugging - How do I debug a WordPress plugin? - Stack Overflow, accessed June 9, 2025, https://stackoverflow.com/questions/14541989/how-do-i-debug-a-wordpress-plugin
26. Custom WordPress Admin Settings Using Gutenberg - ShopWP, accessed June 9,

2025,
https://wpshop.io/blog/wp-shopify-3-0-progress-update-2-admin-settings/