

Extensions

- [Core concepts](#)
- [Get started building extensions](#)
- [Review guidelines](#)
- [Settings APIs and admin pages](#)
- [Extension Guidelines](#)

Core concepts

This section covers the fundamental principles, best practices, and essential knowledge you need to develop robust, maintainable WooCommerce extensions.

Learn about everything from basic setup and architecture to advanced development patterns. Whether you're building your first WooCommerce extension or maintaining existing ones, these guides will help you follow best practices and create high-quality code.

Getting started

[Check if WooCommerce is active](#) to learn the proper way to ensure WooCommerce is installed and active before your code runs. This prevents errors and ensures your extension works reliably. You'll also want to understand the [core WooCommerce classes](#) and how to work with them, from the main `WooCommerce` class to `WC_Product`, `WC_Customer`, and `WC_Cart`.

Internal code boundaries

WooCommerce contains both public APIs intended for extension use and internal infrastructure code. **Extensions must not use internal code**, as backwards compatibility between WooCommerce releases is not guaranteed:

- **Internal namespace:** All classes under `Automattic\WooCommerce\Internal` are internal infrastructure.
- **@internal annotations:** Code entities (classes, methods, hooks) marked with `@internal` in their docblocks.

Using internal code may cause your extension to break when merchants update WooCommerce. See the [extension development best practices](#) and the [Internal namespace documentation](#) for details.

Development patterns

[Adding actions and filters](#) to master the art of extending WooCommerce through hooks. Learn when and how to add actions and filters, following WordPress and WooCommerce standards. For long-term success, discover strategies for [writing maintainable code](#) and establishing update processes that keep your extensions current and secure. You'll also need to [manage deactivation and uninstallation](#) to ensure your extension cleans up properly when deactivated or uninstalled, including scheduled actions, admin notes, and tasks.

Plugin structure and standards

See the [example header plugin comment](#) format for your extension's main plugin file header, including all required metadata. You'll also want to learn the standard [changelog format](#) for documenting changes in your extension's changelog file, and understand the [WooCommerce plugin API callback](#) for proper integration with WooCommerce's plugin API for seamless functionality.

Get started building extensions

This section provides guides and resources for building, testing, and distributing WooCommerce extensions.

Important: Internal vs public code

Not all WooCommerce code is intended for use by extensions. Classes in the `Automattic\WooCommerce\Internal` namespace and code marked with `@internal` are for WooCommerce core use only: backwards compatibility between WooCommerce releases is not guaranteed and your extension may break if you use them. See the [extension development best practices](#) and the [Internal namespace documentation](#).

Getting started

- [Design a simple extension](#) - Learn extension architecture and best practices
- [Build your first extension](#) - Create your first WooCommerce extension
- [Core concepts](#) - Master fundamental concepts like plugin headers, lifecycle management, and security

Submit to the WooCommerce Marketplace

Join the WooCommerce Marketplace and get your extension in front of 3.6M+ active stores worldwide.

Learn more about [why extension developers are choosing the WooCommerce Marketplace](#) and [submit your extension](#)

Quality Insights Toolkit (QIT)

Available to all developers with a WooCommerce.com vendor profile

QIT (Quality Insights Toolkit) is a testing platform developed by WooCommerce for plugins and themes. It allows developers to quickly run a variety of managed tests out-of-the-box, as well as integrate their own custom E2E tests to ensure their extensions are reliable, secure, and compatible.

Key features

- **Managed test suites:** Run pre-configured end-to-end tests, activation tests, security scans, PHPStan analysis, API tests, and more
- **Custom E2E testing:** Write and run your own Playwright-based E2E tests directly through QIT
- **Continuous quality checks:** Seamlessly integrate QIT into your development workflows via CLI, GitHub Actions, and more
- **Marketplace integration:** Currently in closed beta for extensions listed on the WooCommerce Marketplace

[Learn more about QIT](#)

Development tools

- [Extension scaffolds](#) - Learn how to scaffold new extensions with our [create-woo-extension](#) package.
- [WooCommerce CLI](#) - Command-line tools for WooCommerce development

Review guidelines

Reviews are an integral part of the online shopping experience, and people installing software pay attention to them. Prospective users of your extensions will likely consider average ratings when making software choices.

Many of today's most popular online review platforms - from Yelp business reviews, to Amazon product reviews - have a range of opinion that can be polarized, with many extremely positive and/or negative reviews, and fewer moderate opinions. This creates a "J-shaped" distribution of reviews that isn't as accurate or as helpful as could be.

WooCommerce.com and WordPress.org both feature reviews heavily, and competing extensions having a higher rating likely have the edge in user choice.

Primary considerations around reviews

Requesting more reviews for a extension with major issues will not generate good reviews, and analyzing existing reviews will help surface areas to address before soliciting reviews.

It is extremely rare for users of WordPress plugins to leave reviews organically (.2% of users for WordPress.org leave reviews), which means that there's an untapped market of 99.8% of users of the average plugin.

These plugins are competing with other plugins on the same search terms in the WordPress.org plugin directory, and ratings are a large factor in the ranking algorithm. This is not usually a factor to the same extent on the WooCommerce Marketplace. For instance, WooCommerce's PayPal extension directly competes on all possible keywords with other PayPal extensions on the WordPress.org repository, while it does not compete with other PayPal payments extensions on the Marketplace.

Settings APIs and admin pages

Managing your settings, admin pages, and more is possible through the Settings APIs. The docs in this section cover what the Settings APIs and admin pages system offers in general. This guide goes further to map all of the various approaches you have available, some of the scenarios they cover, and the trade-offs of each approach.

There are also code examples for your reference before you commit to an implementation path. However, reading through the approach you're interested in will give you greater scope to understand and apply the principles of each.

Approaches at a glance

The Settings API and admin pages section has six approaches, outlined in the table. The four core concepts (Settings API, settings pages, settings tab sections, and admin pages) cover most settings and admin placement decisions, while the remaining approaches cover specialized integrations and navigation shortcuts.

Approach	Use when	Requires
Settings API	Defining settings for a configurable component, such as a payment gateway or shipping method	PHP
Settings pages	Your extension needs a full tab under WooCommerce > Settings with one or more sections	PHP
Settings tab sections	Your settings belong under an existing WooCommerce tab rather than a new page	PHP
Admin pages	Your extension needs a standalone page outside WooCommerce's settings structure	PHP or JavaScript
Integration settings	Your extension connects to a third-party service and needs a page under the Integrations tab	PHP

Approach	Use when	Requires
Store management links	Surfacing quick-access links on the WooCommerce home screen	JavaScript

Settings API

`WC_Settings_API` provides field definition, rendering, loading, and saving for configurable WooCommerce components. Plugin developers usually inherit it through `WC_Payment_Gateway`, `WC_Shipping_Method`, or `WC_Integration` rather than extending it directly.

Use the [Payment Gateway API](#), [Shipping Method API](#), or [integration settings](#) guide for a complete implementation.

Settings pages

Use `WC_Settings_Page` when your extension needs a full tab under **WooCommerce > Settings**. A settings page class registers the tab, renders one or more sections, defines fields with WooCommerce's settings array format, and lets WooCommerce handle saving through the existing settings form.

The main trade-off is navigation weight. If your extension only needs a few settings that clearly belong under an existing WooCommerce settings tab, add a section to that tab instead of creating a new top-level tab. If the extension needs an interface outside WooCommerce settings entirely, use admin page registration.

```
final class My_Plugin_Settings_Page extends WC_Settings_Page {
    public function __construct() {
        $this->id      = 'my_plugin';
        $this->label    = __( 'My plugin', 'my-plugin' );

        parent::__construct();
    }

    protected function get_settings_for_default_section() {
        return array(
            array(
                'title' => __( 'My plugin settings', 'my-plugin' ),
                'type'  => 'title',
            )
        );
    }
}
```

```

        'id'      => 'my_plugin_options',
    ),
    array(
        'title'    => __( 'Enable feature', 'my-plugin' ),
        'id'       => 'my_plugin_enabled',
        'type'     => 'checkbox',
        'default'  => 'no',
    ),
    array(
        'type' => 'sectionend',
        'id'   => 'my_plugin_options',
    ),
);

}

add_filter(
    'woocommerce_get_settings_pages',
    function( $settings_pages ) {
        $settings_pages[] = new My_Plugin_Settings_Page();
        return $settings_pages;
    }
);

```

Settings tab sections

Rather than creating a whole new page, you can add a section beneath an existing WooCommerce settings tab using two filters:

- `woocommerce_get_sections_{tab}` registers the section.
- `woocommerce_get_settings_{tab}` supplies its fields.

The `{tab}` portion of each filter corresponds to the settings page ID for the tab you want to extend. For instance, `products` targets the **Products** tab, while `account` targets **Accounts & Privacy**.

Placing settings under an existing tab keeps the admin area organized and means merchants find your extension's options in an understandable context. The limit is that linking to your section from documentation or onboarding flows requires a URL with both a tab and a section parameter rather than a page URL you control.

```

add_filter(
    'woocommerce_get_sections_products',
    function( $sections ) {
        $sections['my_extension'] = __( 'My Extension', 'my-extension' );
        return $sections;
    }
);

add_filter(
    'woocommerce_get_settings_products',
    function( $settings, $current_section ) {
        if ( 'my_extension' === $current_section ) {
            return array(
                array(
                    'title' => __( 'My Extension', 'my-extension' ),
                    'type'   => 'title',
                    'id'     => 'my_extension',
                ),
                array(
                    'title' => __( 'Enable feature', 'my-extension' ),
                    'type'   => 'checkbox',
                    'id'     => 'my_extension_enabled',
                ),
                array(
                    'type' => 'sectionend',
                    'id'   => 'my_extension',
                ),
            );
        }

        return $settings;
    },
    10,
    2
);

```

Admin page registration

When your extension needs a page outside WooCommerce's settings structure, you register it with the `PageController`. This attaches the WooCommerce Admin header and activity panel to the page and WooCommerce provides two registration paths:

- `wc_admin_connect_page()` for PHP-powered pages.
- `wc_admin_register_page()` for React-powered pages.

PHP registration connects an existing admin page to the WooCommerce Admin shell. This will suit extensions that already render an interface with PHP. React registration creates a menu entry and renders a component you supply, which gives you a component-driven interface but requires a JavaScript build step.

Both paths are for extensions with an interface that's too distinct to fit within WooCommerce's settings tabs. For example, this could be a reporting dashboard, a product import tool, or any page with its own layout.

```
// PHP-powered page.
wc_admin_connect_page(
    array(
        'id'          => 'my-extension-page',
        'screen_id' => 'my-extension-page',
        'title'       => array( 'My Extension', 'Settings' ),
        'path'        => add_query_arg( 'page', 'my-extension', 'admin.php' ),
    )
);
```

```
// React-powered page.
import { addFilter } from '@wordpress/hooks';

const MyPage = () => <h1>My Extension</h1>;

addFilter( 'woocommerce_admin_pages_list', 'my-extension', ( pages ) => {
    pages.push( {
        container: MyPage,
        path: '/my-extension',
        breadcrumbs: [ 'My Extension' ],
    } );

    return pages;
} );
```

Integration settings

`WC_Integration` extends `WC_Settings_API` and creates a settings page under **WooCommerce > Settings > Integrations**. It also handles data saving and sanitization for you. You should use it when your extension connects to an external service and a dedicated page under the **Integrations** tab is the right home for its settings.

The main trade-off is placement: if your settings belong under an existing WooCommerce tab or on a standalone page, this is not the path. `WC_Integration` is great for fields such as API keys or toggle switches because you get saving and sanitization without writing that logic yourself.

```
class My_Integration extends WC_Integration {
    public function __construct() {
        $this->id = 'my-integration';
        $this->method_title = __( 'My Integration', 'my-integration' );
        $this->method_description = __( 'Connect to My Service.', 'my-integration' );
        $this->init_form_fields();
        $this->init_settings();
        add_action(
            'woocommerce_update_options_integration_' . $this->id,
            array( $this, 'process_admin_options' )
        );
    }

    public function init_form_fields() {
        $this->form_fields = array(
            'api_key' => array(
                'title' => __( 'API Key', 'my-integration' ),
                'type' => 'text',
            ),
        );
    }

    add_filter(
        'woocommerce_integrations',
        function( $integrations ) {
            $integrations[] = 'My_Integration';
            return $integrations;
        }
    );
}
```

```
    }  
  );
```

Store management links

The WooCommerce home screen includes a store management dashboard that displays quick-access links and statistics for merchants. You can add your own link there using the `woocommerce_admin_homescreen_quicklinks` JavaScript filter.

However, two constraints apply:

- Links must point to pages within WooCommerce as external URLs are not supported.
- All extension-added links appear under a fixed **Extensions** category and custom categories are not available.

You enqueue your script using `admin_enqueue_scripts`, with a dependency on `wp-hooks` and a priority higher than 15 to ensure it runs before the section renders.

Note that this approach doesn't *configure* settings as its purpose is discoverability for merchants who use the WooCommerce home screen as a starting point for day-to-day tasks.

```
import { megaphone } from '@wordpress/icons';  
import { addFilter } from '@wordpress/hooks';  
  
addFilter( 'woocommerce_admin_homescreen_quicklinks', 'my-extension', ( quickLinks ) => {  
  return [  
    ...quickLinks,  
    {  
      title: 'My Extension',  
      href: 'link/to/my-extension',  
      icon: megaphone,  
    },  
  ];  
} );
```

Next steps

With the trade-offs and code examples for each approach as a reference, you can work through the full documentation for the one your extension needs.

1. [Settings API](#). Core reference for the `WC_Settings_API` class.
2. [Adding a settings page](#). Creating a full tab under **WooCommerce > Settings**.
3. [Adding a section to a settings tab](#). Adding a section beneath an existing WooCommerce tab.
4. [Working with WooCommerce admin pages](#). Registering PHP and React admin pages.
5. [Implementing settings with `WC\_Integration`](#). How to use the `WC_Integration` class for third-party service connections.
6. [Adding store management links](#). Navigation shortcuts on the WooCommerce home screen.

Extension Guidelines

This section covers general guidelines, and best practices to follow in order to ensure your product experience aligns with WooCommerce for ease of use, seamless integration, and strong adoption.

We strongly recommend you review the current [WooCommerce setup experience](#) to get familiar with the user experience and taxonomy.

We also recommend you review the [WordPress core guidelines](#) to ensure your product isn't breaking any rules, and review [this helpful resource](#) on content style.

General

Use existing WordPress/WooCommerce UI, built in components (text fields, checkboxes, etc) and existing menu structures.

Plugins which draw on WordPress' core design aesthetic will benefit from future updates to this design as WordPress continues to evolve. If you need to make an exception for your product, be prepared to provide a valid use case.

- [WordPress Components library](#)
- [Figma for WordPress](#) | ([WordPress Design Library Figma](#))
- [WooCommerce Component Library](#)

Component Library - Storybook

“ Storybook is an open source tool for developing UI components in isolation for React, React Native and more. It makes building stunning UIs organized and efficient.

The WooCommerce repository also includes [Storybook](#) integration that allows testing and developing in a WooCommerce-agnostic context. This is very helpful for developing reusable components and trying generic JavaScript modules without any backend dependency.

You can launch Storybook by running `pnpm --filter=@woocommerce/storybook watch:build` locally. It will open in your browser automatically.

You can also test Storybook for the current `trunk` branch on GitHub Pages:

<https://woocommerce.github.io/woocommerce>