ROOTED DEV GUIDES

PHP for WordPress Developers - Quick Reference

A compact PHP and WordPress reference for the patterns you will reach for constantly while building themes and plugins.

FREE COMPANION RESOURCE

Companion to Book 5 - PHP for WordPress Developers From the Root Up

+ Refresh core PHP syntax without leaving the WordPress context.
+ Keep common hook, sanitization, escaping, capability, query, and debugging patterns handy.
+ Use the safety checklist before accepting input or changing WordPress data.

How to use this: Work through the pages, mark what applies to you, and keep the PDF as a practical reference. This resource is intentionally concise; the full book goes deeper into the why, how, and real-world implementation.

Learn WordPress from the root up.

1. PHP Building Blocks

These are the everyday language pieces behind most WordPress theme and plugin code.

Variables & arrays

```
$title = 'Hello';
$count = 3;
$active = true;
$items = array( 'one', 'two' );
$settings = array( 'enabled' => true, 'limit' => 10 );
```

Use descriptive names. WordPress codebases often favor arrays for configuration and API arguments.

Conditionals

```
if ( $active && $count > 0 ) {
// Do something.
} elseif ( $count === 0 ) {
// Empty state.
} else {
// Fallback.
}
```

Prefer strict comparisons when the type matters.

Loops

```
foreach ( $items as $item ) {
echo esc_html( $item );
}
```

Escape dynamic output at the point it is rendered.

Functions

```
function rooted_format_label( $value ) {
$value = sanitize_text_field( $value );
return ucwords( $value );
}
```

Give functions one clear responsibility and use a unique prefix or namespace in distributable code.

2. WordPress Hooks

Actions let you run code at a point in WordPress. Filters let you receive a value, modify it, and return it.

Action example

```
add_action( 'init', 'rooted_register_feature' );

function rooted_register_feature() {
// Registration code.
}
```

Filter example

```
add_filter( 'the_content', 'rooted_add_note' );

function rooted_add_note( $content ) {
return $content . '<p>Extra note.</p>';
}
```

Hook reminders

[ ] Use the correct hook for the lifecycle moment you need.

[ ] Match accepted argument counts when registering callbacks.

[ ] Filters must return the filtered value.

[ ] Avoid expensive work on hooks that run on every request unless it is necessary.

[ ] Remove or replace hooks only when you understand what registered them and when.

3. Input, Output & Permissions

A useful mental model: validate what is allowed, sanitize what you accept, authorize who may act, and escape what you output.

Sanitize text input

```
$name = isset( $_POST['name'] ) ? sanitize_text_field( wp_unslash( $_POST['name'] ) ) : '';
```

Sanitization does not replace validation. Check required formats and allowed values too.

Escape HTML output

```
echo esc_html( $name );
echo esc_attr( $value );
echo esc_url( $url );
```

Use the escaping function that matches the output context.

Capability check

```
if ( ! current_user_can( 'manage_options' ) ) {
wp_die( esc_html__( 'You are not allowed to do that.', 'rooted-example' ) );
}
```

Permissions should be checked on the server for privileged actions.

Nonce check

```
check_admin_referer( 'rooted_save_settings' );
```

A nonce helps protect request intent. It is not an authorization system, so pair it with capability checks.

4. Common WordPress Data Patterns

Read an option

```
$settings = get_option( 'rooted_settings', array() );
```

Always provide or handle a sensible default.

Update an option

```
update_option( 'rooted_settings', $settings );
```

Store only the data the plugin actually needs.

WP_Query

```
$query = new WP_Query( array(
'post_type' => 'post',
'posts_per_page' => 10,
'post_status' => 'publish',
) );
```

Reset post data after a custom loop when you use setup_postdata().

Prepared database query

```
global $wpdb;
$row = $wpdb->get_row(
$wpdb->prepare(
"SELECT * FROM {$wpdb->prefix}example WHERE id = %d",
$id
)
);
```

Prefer WordPress data APIs first. When direct SQL is justified, prepare variable values correctly.

REST response

```
return new WP_REST_Response( array( 'success' => true ), 200 );
```

REST routes should define permission_callback and validate request arguments.

5. Debugging & Safe Development

Use a staging or local environment whenever you are learning, refactoring, or changing data.

Debugging checklist

[ ] Enable WP_DEBUG in development and review PHP notices/warnings instead of hiding them.

[ ] Use error_log() for targeted server-side debugging rather than printing sensitive data to visitors.

[ ] Check the browser console for JavaScript errors.

[ ] Check the Network panel when AJAX or REST behavior is unexpected.

[ ] Reproduce the problem with a default theme or minimal plugin set when isolating conflicts.

Before changing data

[ ] Confirm the current user has permission.

[ ] Verify request intent for state-changing operations.

[ ] Validate the expected type and allowed values.

[ ] Sanitize incoming values at the boundary.

[ ] Use the right WordPress API for the data being changed.

[ ] Escape dynamic output when it is rendered.

[ ] Keep a backup before running bulk or destructive operations.

Remember the order
INPUT - Validate -> sanitize -> authorize/verify intent as required -> process/store.
OUTPUT - Retrieve/process -> escape for the exact output context -> render.

GO DEEPER WITH THE FULL GUIDE

Book 5 - PHP for WordPress Developers From the Root Up

The full guide builds these patterns from the ground up and explains how PHP behaves inside real WordPress development, including functions, arrays, classes, hooks, data, security, debugging, and practical plugin/theme examples.

rooteddevguides.com