Hands-On WordPress Meetup Activity

Debug Like a Pro: WordPress Developer Debugging Tools

Facilitator Guide

Overview

Guessing at bugs wastes time. This 60-minute activity equips developers with four complementary debugging tools — WP_DEBUG and error logs, Query Monitor, browser DevTools, and Xdebug — and teaches them to match the right tool to each type of problem. Parts 1–3 run entirely in WordPress Playground. Part 4 is a facilitator-led Xdebug demonstration using the Playground CLI.

Topic	Debug Like a Pro: WordPress Developer Debugging Tools
Duration	60 minutes
Audience	WordPress developers: plugin/theme builders, custom site developers
Prerequisite	Familiarity with WordPress themes, plugins, and the admin dashboard; basic PHP awareness. Part 4 facilitator demo requires Node.js 18+ installed locally.
Environment	WordPress Playground (browser-based) for Parts 1–3; local CLI for Part 4 facilitator demo
Materials	Device with a modern browser (Chrome or Firefox recommended) and internet access

⚠ **Version note:** Written for WordPress 7.0. Verify steps against the current release before running. Release notes: wordpress.org/news | Core development blog: make.wordpress.org/core

WordPress 7.0 changes in this activity: The default admin color scheme is now “Modern” — a lighter, higher-contrast palette. The admin will look different from screenshots taken on older installs. The underlying menu structure and navigation are unchanged.The Plugin File Editor (used in Part 1) now runs CodeMirror v5. The editor looks slightly updated but the editing workflow is identical.New in 7.0: wp_trigger_error() can be hooked even when WP_DEBUG is off (Trac §60886). Plugins can now

intercept and handle PHP errors without requiring debug mode to be enabled — a useful point to raise during the Part 1 WP_DEBUG discussion.

Learning Objectives

By the end of this activity, participants will be able to:

- **Configure —** WP_DEBUG, WP_DEBUG_LOG, and WP_DEBUG_DISPLAY in wp-config.php and read the resulting debug.log.
- **Interpret —** PHP errors, warnings, and notices to identify the file, line, and root cause.
- **Use —** Query Monitor to surface slow database queries, mis-ordered hooks, and HTTP API failures.
- **Diagnose —** JavaScript errors in the block editor and front end using browser DevTools.
- **Explain —** how Xdebug breakpoints work and when to reach for step-debugger over log-based debugging.
- **Select —** the right debugging tool for a given problem type.

Facilitator Setup Notes

Before the session	Verify playground.wordpress.net loads in your browser. Test Query Monitor URL: playground.wordpress.net/?plugin=query-monitor
Part 4 — Xdebug	Parts 1-3 are browser-only. Part 4 requires you to run: npx @wp-playground/cli@latest server --xdebug --experimental-devtools --auto-mount from your terminal. Test this before the session on the presentation machine. Node.js 18+ is required.
Part 4 format	Part 4 is a facilitator-led live demo. Participants watch and ask questions rather than following along independently. This is because Xdebug requires local CLI setup that varies by machine.
Query Monitor note	Query Monitor requires being logged in as Administrator. Playground's default user is admin — this satisfies the requirement.

Part 1 · 10 minutes — WP_DEBUG and Error Logs — Turning the Lights On

Background

By default, WordPress suppresses error output. Enabling WP_DEBUG turns on PHP error reporting. Adding WP_DEBUG_LOG writes errors to wp-content/debug.log instead of displaying them on screen — safe for production-like environments. WP_DEBUG_DISPLAY controls whether errors appear inline on the page.

What participants do

1. Open playground.wordpress.net in a new tab.
2. Open the **Sidebar panel** (panel icon, top-right Playground toolbar). Click the **File browser** tab.
3. Navigate to **wp-config.php** in the site root and click it to open the editor.
4. Find the line that reads ***define('WP_DEBUG', false);*** and replace it with the following three constants:

```
define( 'WP_DEBUG', true );
define( 'WP_DEBUG_LOG', true );
define( 'WP_DEBUG_DISPLAY', false );
```

5. The File Browser saves automatically — no need to click Save. Your changes take effect as soon as you finish editing.
6. Activate Hello Dolly so it runs during page load: go to **Plugins** in the admin sidebar, find **Hello Dolly**, and click **Activate**. (Hello Dolly must be active for its function to fire and generate log output when triggered.)
7. Now trigger a PHP warning. Go to **Tools > Plugin File Editor**. **Note:** a 'Heads up!' warning modal appears on first open — click **I understand** to proceed. Select **Hello Dolly** and add this line anywhere inside the **hello_dolly()** function:

```
echo $undefined_variable;
```

8. Click **Update File**, then reload the admin dashboard. Nothing visible should change (because WP_DEBUG_DISPLAY is false).
9. Return to the File Browser. Navigate to **wp-content > debug.log** and click it. You should see a PHP Warning entry. Read the error format: **type, message, file path, and line number**.

> **Facilitator tip:** The three constants form a complete, safe debug setup: errors are captured in debug.log without leaking to visitors. Emphasize that WP_DEBUG_DISPLAY should be false in any environment visitors might see — it can expose stack traces with file paths and configuration details.
>
> In PHP 8.0+, accessing an undefined variable is a Warning (not a Notice as in PHP 7). Playground uses PHP 8.3, so participants will see: PHP Warning: Undefined variable $undefined_variable in … Warnings without fatal consequences are fine for this demo — the key point is that the same workflow captures Warnings, Notices, and fatal Errors. The log entry format is identical: PHP [type]: [message]

in [file] on line [number].

Alternative: the WP Debugging plugin (wordpress.org/plugins/wp-debugging) can toggle these constants from the admin dashboard without editing wp-config.php — useful on client sites where you do not want to touch config files directly.

Part 2 · 20 minutes — Query Monitor — X-Ray Vision for Every Page Load

Background

Query Monitor adds a developer tools panel to the WordPress admin bar. It surfaces database queries, PHP errors, hook load order, HTTP API calls, enqueued scripts and styles, block editor data, and more — on every page load. No configuration required; install and refresh.

What participants do

1. Load Playground with Query Monitor pre-installed: playground.wordpress.net/?plugin=query-monitor.
2. In the admin bar, click the **performance stats** — the row of numbers showing timing, memory, and query count (for example: ***0.17s 32.7MB 0.05s 45Q***). There is no 'Query Monitor' text label in the admin bar. Clicking those numbers opens the Query Monitor panel at the bottom of the screen.
3. Click the **Database Queries** tab. Examine the list of database queries. Look for any queries **highlighted in red** — these are the slow ones. Check the execution time shown next to each query and note which plugin or component called it.

Version note: Sorting by the Time column was removed in Query Monitor v4 due to an accessibility issue. The plugin author (John Blackbourn) has confirmed the feature will be restored in a future update. Until then, slow queries are identified by red highlighting rather than column sorting. Source: wordpress.org/support/topic/unable-to-sort-db-queries-by-time-after-update/.

4. Click the **Hooks & Actions** tab. The full hook list has 300+ entries. Use the **Filter by hook name** dropdown at the top of the panel to find **init**. Type 'i' in the filter field to cycle to entries beginning with that letter, then narrow further until 'init' is selected. Callbacks registered to that hook are displayed inline — note the order they are called and which components registered them. This is where hook priority conflicts become visible.
5. Look for the **PHP Errors** tab. **Note:** this tab only appears when WordPress has detected PHP errors during the current request — on a clean Playground with no errors, the tab will not be visible. When it does appear, it catches errors and notices and shows them in context with the request that triggered them.

6. Navigate to the front end of the Playground site. The Query Monitor panel is still available in the admin bar. Click **HTTP API Calls** to see any outbound HTTP requests made during page load, including their method, URL, and response code.
7. Return to the admin. **Important:** the Block Editor opens in fullscreen mode by default, which hides the admin bar — and therefore Query Monitor. Before opening a post, exit fullscreen mode: click the ⋮ **Options** menu (top-right of the editor) and uncheck **Fullscreen mode**. Then on the admin dashboard, Query Monitor shows an **Admin Screen** tab (not 'Block Editor'); on the front end it shows a **Blocks** tab. Examine the registered blocks and editor-related data in whichever context applies.

> **Facilitator tip:** Query Monitor is the de facto standard debugging plugin for WordPress developers. It was created by John Blackbourn and is open source, free, and maintained actively.
>
> The Queries tab is the most commonly used: it is where you diagnose N+1 query bugs (a loop that fires a separate database query for each item) and over-querying plugins.
>
> The Hooks tab answers the question: 'Why is my filter not running?' or 'Why is my output appearing twice?' Hook priority and duplicate registrations are visible at a glance.
>
> Query Monitor does not require WP_DEBUG to be enabled. It works independently and always shows live data for the current request.
>
> Free tier: all features. No paid version exists. Available at wordpress.org/plugins/query-monitor.

Part 3 · 10 minutes — Browser DevTools — Catching JavaScript Errors

Background

Many WordPress errors that look like PHP problems are actually JavaScript errors. A broken block editor, a non-saving metabox, or a frontend widget that stops working often has a JS error as the root cause — one that never touches PHP or the error log. Browser DevTools is where those errors surface.

What participants do

1. In Playground, open the Block Editor (Posts > Add New).
2. Open browser DevTools: **F12** (Windows/Linux) or **Cmd+Option+I** (Mac). Click the **Console** tab.
3. To generate a real JS error: in the Playground File Browser, create a new file at **wp-content/plugins/bad-js/bad-js.php** with this content:

```
<?php
/*
 * Plugin Name: Bad JS Demo
 * Description: Demonstrates a JavaScript error.
 */
add_action( 'wp_footer', function() {
    echo '<script>undeclaredFunction();</script>';
```

```
} );
```

4. Go to the Plugins screen and activate **Bad JS Demo**. Visit the front end. In the DevTools Console tab, observe the **ReferenceError: undeclaredFunction is not defined** error.
5. Click the source link in the error to jump to the Sources tab. Because this error comes from an inline ***<script>*** tag injected by PHP, DevTools points to the rendered HTML page — not the PHP file itself. This is normal for inline scripts. For errors in enqueued ***.js*** files, the link points directly to the JavaScript source file and line number.
6. Click the **Network** tab in DevTools. Reload the front end. Filter by **Fetch/XHR**. **Note:** a standard front-end page typically has few or no Fetch/XHR entries — REST API traffic is more visible when working in the block editor admin. A red entry in either context indicates a failed request — click it and check the **Response** tab for the WordPress error message.

> **Facilitator tip:** The Console is the first place to check when something in the editor 'just stopped working.' Block editor errors, REST API authentication failures, and conflicting script registrations all appear here before they appear anywhere in PHP logs.
>
> Reading a stack trace: the top line is where the error was thrown; scroll down to find the first line of your own code in the trace — that is usually where to fix it.
>
> Network tab tip: filter by '400' or '500' status codes to find failed REST API requests quickly. A 403 on a REST endpoint often means a nonce mismatch or capabilities issue.
>
> WP 7.0 iframed editor note: WordPress 7.0 enforces an iframed block editor for posts using Block API v3+ (which is all core blocks). If participants open DevTools while editing a post, block-level errors appear inside an iframe execution context. They will see a frame selector dropdown at the top of the Console — participants need to choose 'gutenberg-iframe' (or similar) to see errors thrown by block JavaScript. Errors on the outer admin page (enqueued scripts, PHP-injected JS) still appear in the top-level frame as normal.

Part 4 · 15 minutes — Xdebug in Playground — Step-Debugging PHP [Facilitator Demo]

Format note

This part is a facilitator-led live demonstration. Participants observe and ask questions rather than following along independently, because Xdebug requires local CLI setup (Node.js + the Playground CLI) that varies by machine. A setup guide is included at the end of this section for participants who want to try it after the session.

Background

Log-based debugging (WP_DEBUG + echo statements) tells you what happened. Xdebug tells you what is happening right now, step by step, with every variable's value visible at each line. It eliminates guesswork for logic bugs where the output is wrong but no error is thrown.

Facilitator: what to demonstrate

1. In your terminal, run:

```
npx @wp-playground/cli@latest server --xdebug --experimental-devtools --
auto-mount
```

2. The terminal prints a CDP URL: ***devtools://devtools/bundled/inspector.html?ws=127.0.0.1:9229***. Open this URL in Chrome. Chrome DevTools opens connected to the PHP runtime.
3. In the Sources tab, navigate to a plugin file (e.g., a simple plugin you created in your working directory). Click a line number to set a breakpoint.
4. Trigger the code path in the browser (load a page or perform an action). Execution pauses at the breakpoint.
5. Walk participants through the debugger panel: **Call Stack** (how we got here), **Scope / Variables** (current variable values), **Step Over / Step Into / Step Out** buttons.
6. Change a variable's value in the console pane and step forward to show how the output changes. Key message: you are watching the code execute in real time.

> **Facilitator tip:** Focus on the conceptual shift: from 'add var_dump and reload' to 'set a breakpoint and inspect'. Xdebug is not faster for simple issues — it earns its setup cost on complex logic bugs where var_dump would require 50 iterations to narrow down.
>
> VS Code / PhpStorm integration: mention that with the --experimental-unsafe-ide-integration=vscode flag, breakpoints can be set directly in the VS Code editor. This is the most practical workflow for daily development.
>
> This is an experimental feature. Participants should expect rough edges and are encouraged to share feedback in the #playground Slack channel.
>
> Documentation: wordpress.github.io/wordpress-playground/developers/xdebug/getting-started/

Post-session setup guide for participants

Requirements: Node.js 18+, Chrome browser.

```
# Install and run
npx @wp-playground/cli@latest server --xdebug --experimental-devtools

# With VS Code IDE integration
npx @wp-playground/cli@latest server --xdebug \
    --experimental-unsafe-ide-integration=vscode --auto-mount
```

Full guide: wordpress.github.io/wordpress-playground/developers/xdebug/getting-started/

Debrief — 5 Minutes

1. Given a slow page load, a broken editor, and a plugin error on update: which tool would you reach for first for each?
2. Before today, what was your debugging process? Where did you waste the most time?
3. What would it take for you to set up Xdebug in your daily development workflow?

Reference card — which tool for which problem:

Problem type	Best first tool
PHP fatal error / notice / warning	WP_DEBUG + debug.log
Slow page load / N+1 queries / hook conflict	Query Monitor
Block editor broken / REST API failure	Browser DevTools Console + Network
Logic bug with no visible error	Xdebug breakpoints

Sources

Source	URL	Notes
Debugging in WordPress — Advanced Administration Handbook	https://developer.wordpress.org/advanced-administration/debug/debug-wordpress/	Canonical reference for WP_DEBUG, WP_DEBUG_LOG, WP_DEBUG_DISPLAY. Last updated July 2025.
Query Monitor — WordPress.org	https://wordpress.org/plugins/query-monitor/	Free, open-source developer tools panel. All features free.
Useful Debugging Plugins — Learn WordPress	https://learn.wordpress.org/lesson/useful-debugging-plugins/	Official Learn WordPress lesson covering Query Monitor and Debug Bar.
Helper Plugins — Plugin Handbook	https://developer.wordpress.org/plugins/developer-tools/helper-plugins/	Official Plugin Handbook section on developer debugging tools.
Using Your Browser to Diagnose JavaScript Errors — Advanced Administration Handbook	https://developer.wordpress.org/advanced-administration/debug/debug-javascript/	Official guide to browser console debugging for WordPress.
Debugging with Xdebug in WordPress Playground	https://wordpress.github.io/wordpress-playground/developers/xdebug/introduction/	Official Playground documentation for Xdebug integration.
Getting Started with Xdebug — WordPress Playground	https://wordpress.github.io/wordpress-playground/developers/xdebug/getting-started/	Step-by-step guide for enabling Xdebug via @wp-playground/cli.

Debugging with Xdebug is Now Available in WordPress Playground — Make WordPress Playground	https://make.wordpress.org/playground/2025/11/24/debugging-with-xdebug-is-now-available-in-wordpress-playground/	Official announcement confirming Xdebug availability. November 2025.