WordPress Intermediate/Advanced

A Complete 10-Chapter Web Platforms Course

Topics covered:

Theme anatomy & the template hierarchy · the Loop & WP_Query

Custom post types & taxonomies · plugin development (actions & filters)

The REST API & headless WordPress · proper script/style enqueuing

Security hardening (nonces, capabilities, escaping, $wpdb->prepare())

Performance & caching mapped to Core Web Vitals

Written for the developer — real PHP throughout

Companion course: WordPress Fundamentals covers the site-owner side

Capstone: a real theme + custom post type + secured contact form

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Custom Theme Development, Getting Started

WordPress Intermediate/Advanced

Chapter 1 · Custom Theme Development, Getting Started

WordPress Fundamentals 5 treated a theme as a black box — something you install and activate, never something you build. This course opens that box, starting with the two files WordPress absolutely requires, and the two more that, by strong convention, essentially every real theme includes.

The Two Files WordPress Actually Requires

A folder inside `wp-content/themes/` only becomes a real, recognized WordPress theme once it contains exactly two specific files.

style.css — Required, and Genuinely Unusual

Every theme's `style.css` must begin with a specially-formatted comment block that WordPress itself parses — this is what actually registers the folder as a theme in the first place and makes it appear in the dashboard's theme browser at all.

```
/*
Theme Name: My First Theme
Theme URI: https://example.com/my-first-theme
Author: Your Name
Description: A minimal custom theme built from scratch.
Version: 1.0
License: GNU General Public License v2 or later
Text Domain: my-first-theme
*/
```

THIS COMMENT BLOCK ISN'T DECORATIVE

Without this exact header, WordPress won't recognize the folder as a theme at all — it simply won't appear in **Appearance → Themes**, no matter how much other CSS the file contains below it. The comment is doing real, functional work, not just documenting the theme for humans.

index.php — Required, the Universal Fallback

`index.php` is the one template guaranteed to exist and be used if no more specific template file is found for a given page — the fallback of last resort. WordPress Intermediate/Advanced 2 covers the full system

determining exactly which template file gets chosen for which URL; this chapter only needs the fact that `index.php` is always the final, catch-all option.

Two More Files Nearly Every Real Theme Includes

Not strictly required by WordPress core, but close to universal by convention: `header.php` and `footer.php`, pulled into other templates via `get_header()` and `get_footer()`.

```
<!-- inside index.php -->
<?php get_header(); ?>

  <main>
    <p>Page content will go here.</p>
  </main>

<?php get_footer(); ?>
```

A GENUINE, COMMON GOTCHA

`header.php` must call `wp_head()` just before its closing `</head>` tag, and `footer.php` must call `wp_footer()` just before its closing `</body>` tag. These two hooks are how WordPress core and every installed plugin inject their own scripts and styles into the page — a theme missing either one will cause plugins to silently, confusingly fail to work, with no obvious error pointing back to the actual cause.

```
<!-- header.php, minimal example -->
<!DOCTYPE html>
<html <?php language_attributes(); ?>>
<head>
  <meta charset="<?php bloginfo( 'charset' ); ?>">
  <title><?php bloginfo( 'name' ); ?></title>
  <?php wp_head(); ?>
</head>
<body <?php body_class(); ?>>
```

```
<!-- footer.php, minimal example -->
  <?php wp_footer(); ?>
</body>
</html>
```

What This Actually Reveals About WordPress Fundamentals 5

Every theme WordPress Fundamentals 5 covered installing and activating was, underneath, exactly this: a folder containing at minimum a correctly-formatted `style.css` and an `index.php`, almost certainly alongside `header.php` and `footer.php` — plus, typically, a great deal more template files and a `functions.php` covered properly once WordPress Intermediate/Advanced 7 reaches proper script/style enqueuing.

Hands-On Exercises

EXERCISE 1

A developer creates a folder with a fully-styled `style.css` and a working `index.php`, but forgets the special comment header block. Explain what happens when they check the WordPress dashboard.

View solution

EXERCISE 2

A theme's `header.php` is missing its call to `wp_head()`. A plugin that's supposed to load its own CSS on every page stops working, with no visible error message anywhere. Explain exactly why.

View solution

EXERCISE 3

Write a minimal, valid `style.css` header block for a theme named "Portfolio Grid," and explain what would happen if this file didn't exist at all in an otherwise complete theme folder.

View solution

CHAPTER 1 QUICK REFERENCE

- **style.css** — required; its special comment header block is what registers the folder as a theme at all
- **index.php** — required; the universal fallback template used when no more specific template exists
- **header.php** / **footer.php** — not strictly required, but near-universal by convention, pulled in via get_header()/get_footer()
- **wp_head()** / **wp_footer()** — must be called in header.php/footer.php respectively, or plugin scripts/styles silently fail to load
- Every theme WordPress Fundamentals 5 installed was exactly this structure underneath, typically with many more template files and functions.php

- **Next chapter:** The WordPress Template Hierarchy

CHAPTER 2 OF 10

The WordPress Template Hierarchy

WordPress Intermediate/Advanced

Chapter 2 · The WordPress Template Hierarchy

Chapter 1 named `index.php` as the universal fallback template without explaining how WordPress decides whether to use something more specific first. This chapter covers that full decision process — and it's genuinely just PHP underneath, exactly the kind of logic *PHP Fundamentals* already prepared you to read.

The Core Idea

For any given URL, WordPress builds an ordered list of candidate template filenames — most specific first, most generic last — and uses the **first file that actually exists** in the active theme. If none of the more specific candidates exist, the search always ends at `index.php`, exactly as Chapter 1 described.

> **THIS IS GENUINELY JUST PHP, NOT MAGIC**
>
> Mechanically, WordPress is doing nothing more exotic than building a string, checking whether a file with that name exists (the same kind of file-existence check any PHP developer already knows how to write), and including the first one found. The "hierarchy" is really just an ordered list your code could, in principle, loop through by hand.

The Hierarchy for a Single Blog Post

CHECKED, IN ORDER	EXAMPLE FILENAME
1. Post-type + slug specific	`single-post-my-post-title.php`
2. Post-type specific	`single-post.php`
3. Generic single	`single.php`
4. Generic singular (posts and pages both)	`singular.php`
5. Universal fallback	`index.php`

The Hierarchy for a Page

CHECKED, IN ORDER	EXAMPLE FILENAME
1. A custom template explicitly assigned to that Page	(whichever file was chosen in the editor)
2. Slug specific	`page-about.php`
3. Page ID specific	`page-42.php`
4. Generic page	`page.php`
5. Generic singular	`singular.php`
6. Universal fallback	`index.php`

Archives — Category, Blog Listing, and 404

PAGE TYPE	CHECKED, IN ORDER
Category archive	`category-{slug}.php` → `category-{id}.php` → `category.php` → `archive.php` → `index.php`
Main blog listing	`home.php` → `index.php`
404 error page	`404.php` → `index.php`

A Worked Trace

FOLLOWING ONE REAL URL REQUEST THROUGH THE DECISION PROCESS

A visitor requests a single blog post whose slug is `hello-world`. WordPress checks, in order: does `single-post-hello-world.php` exist in the active theme? No. Does `single-post.php` exist? No. Does `single.php` exist? Yes — that file is used, and the search stops there. If `single.php` hadn't existed either, the search would have continued to `singular.php`, and finally to `index.php` as the last resort.

Template Tags — Pulling in Dynamic Content

Once the right template file is chosen, it needs to actually display the post or page's own content — that's what **template tags** are for: PHP functions called directly inside a template to output dynamic, per-page content.

```
<h1><?php the_title(); ?></h1>
<p>Published on <?php the_date(); ?></p>
<a href="<?php the_permalink(); ?>">Read more</a>
```

`the_content()` — the actual post/page body — deliberately isn't shown here, since it needs to be called inside the Loop, the mechanism WordPress Intermediate/Advanced 3 covers next.

Conditional Tags — One File, Many Behaviors

A minimal theme relying heavily on `index.php` can still behave differently depending on what kind of page is currently being rendered, using conditional tags:

```
<?php if ( is_single() ) : ?>
  <p>You're reading a single post.</p>
<?php elseif ( is_page() ) : ?>
  <p>You're viewing a page.</p>
<?php elseif ( is_category() ) : ?>
  <p>You're browsing a category archive.</p>
<?php endif; ?>
```

WHY A REAL THEME STILL SPLITS INTO MANY FILES

Conditional tags could, in principle, let a single `index.php` handle every page type through nothing but if/elseif branches — but real themes still split into `single.php`, `page.php`, `archive.php`, and so on, because separate files stay far more readable and maintainable than one enormous file branching on every possible page type. The hierarchy exists to make that split convenient, not to discourage it.

Hands-On Exercises

EXERCISE 1

A theme contains only `index.php`, `header.php`, and `footer.php` — no `single.php`, no `page.php`. Explain what template renders a single blog post on this theme, and why.

View solution

EXERCISE 2

A theme has both `single.php` and `singular.php`. Explain which one WordPress actually uses to render a single blog post, and why the other one is effectively unused for that page type.

View solution

EXERCISE 3

Explain why this chapter says the template hierarchy is "genuinely just PHP, not magic," describing the actual underlying mechanism in your own words.

View solution

CHAPTER 2 QUICK REFERENCE

- WordPress checks an ordered list of candidate filenames, most specific first, and uses the first one that exists
- Single post: single-{type}-{slug}.php → single-{type}.php → single.php → singular.php → index.php
- Page: assigned custom template → page-{slug}.php → page-{id}.php → page.php → singular.php → index.php
- Archives/404 each have their own ordered chain, always ending at index.php
- **Template tags** (the_title(), the_permalink(), etc.) output dynamic content; the_content() is deferred to the Loop, next chapter
- **Conditional tags** (is_single(), is_page(), is_category()) let one file behave differently per page type — real themes still split into many files for readability
- **Next chapter:** The Loop & WordPress Core Functions

CHAPTER 3 OF 10

The Loop & WordPress Core Functions

WordPress Intermediate/Advanced

Chapter 3 · The Loop & WordPress Core Functions

WordPress Intermediate/Advanced 2 deliberately left `the_content()` unused because it needs something not yet introduced: the Loop. Every template file in Chapters 1 and 2 has been outputting a single post's data implicitly — this chapter formalizes exactly how, and how to query for posts beyond whatever WordPress already queried automatically.

The Loop, Formalized

The Loop is genuinely just a PHP `while` loop, exactly the kind of construct *PHP Fundamentals* already covers, wrapped around two WordPress-specific functions:

```
<?php if ( have_posts() ) : ?>
  <?php while ( have_posts() ) : the_post(); ?>
    <h2><?php the_title(); ?></h2>
    <?php the_content(); ?>
  <?php endwhile; ?>
<?php endif; ?>
```

- **have_posts()** — returns true while there are still posts remaining in the current query to loop through, exactly what any ordinary PHP `while` condition needs
- **the_post()** — advances to the next post *and* sets up the global post data every template tag (`the_title()`, `the_content()`, `the_permalink()`) reads from, so each iteration's template tags correctly refer to that iteration's own post

Where "The Current Query" Actually Comes From

Before your template file ever runs, WordPress has already built a query based on the requested URL — visiting a category archive automatically queries that category's posts, visiting a single post automatically queries just that one post. This automatic query is why `have_posts()` / `the_post()` simply work in a single/archive template with no query-writing required — the query already happened.

WP_Query — Querying for More Than the Automatic Query Gives You

When a template needs posts *beyond* what WordPress automatically queried — a "related posts" section, a custom list by category — `WP_Query` builds an entirely separate, custom query.

```
<?php
$related = new WP_Query( array(
    'post_type'      => 'post',
    'posts_per_page' => 5,
    'category_name'  => 'news',
) );

if ( $related->have_posts() ) :
    while ( $related->have_posts() ) : $related->the_post();
        the_title();
    endwhile;
    wp_reset_postdata();
endif;
?>
```

WP_RESET_POSTDATA() IS NOT OPTIONAL

Looping a custom `WP_Query` overwrites the same global post data `the_post()` always sets up — after the custom loop finishes, that global state still points at the *last post in the custom query*, not back to the original post the main query was on. Any template tags used after the custom loop, but before calling `wp_reset_postdata()`, will silently reference the wrong post's data — a genuine, common bug with no obvious error message, in the same spirit as WordPress Intermediate/Advanced 1's own `wp_head()` gotcha.

get_posts() — A Simpler Alternative

`get_posts()` runs a custom query too, but returns a plain PHP array of post objects directly, rather than a query object requiring `have_posts()` / `the_post()`.

```
<?php
$recent = get_posts( array(
    'numberposts' => 5,
    'category_name' => 'news',
) );

foreach ( $recent as $post ) : setup_postdata( $post );
    the_title();
endforeach;
wp_reset_postdata();
?>
```

WP_QUERY	GET_POSTS()
Returns a full query object with pagination support	Returns a plain array — no built-in pagination
Loop with have_posts()/the_post(), reset with wp_reset_postdata()	Loop with an ordinary foreach; still needs setup_postdata()/wp_reset_postdata() if using template tags
The right choice for anything needing pagination or deeper query control	The right choice for a quick, simple list with no pagination needed

Hands-On Exercises

EXERCISE 1

A developer adds a "Related Posts" section using WP_Query near the bottom of a single-post template, but forgets to call wp_reset_postdata() afterward. The page footer, which calls the_title() expecting the original post's title, shows the wrong title instead. Explain exactly why.

View solution

EXERCISE 2

Explain why have_posts() and the_post() "just work" on an ordinary single.php template with no query-writing code anywhere in that file.

View solution

EXERCISE 3

A developer needs a paginated list of posts filtered by category, with "next page" / "previous page" navigation. Explain whether WP_Query or get_posts() is the better tool here, and why, using this chapter's own comparison.

View solution

CHAPTER 3 QUICK REFERENCE

- The Loop is an ordinary PHP while loop wrapped around have_posts() (the condition) and the_post() (advances + sets up per-iteration global post data)
- WordPress automatically builds "the current query" from the requested URL before the template even runs
- **WP_Query** — a custom query object with pagination support, looped with have_posts()/the_post(), always paired with wp_reset_postdata() afterward
- **get_posts()** — a simpler function returning a plain array, looped with foreach, no built-in pagination

- Forgetting wp_reset_postdata() leaves global post data pointing at the wrong post — a genuine, silent bug
- **Next chapter:** Custom Post Types & Taxonomies

CHAPTER 4 OF 10

Custom Post Types & Taxonomies

WordPress Intermediate/Advanced

Chapter 4 · Custom Post Types & Taxonomies

WordPress Fundamentals 3 made a promise: since Posts and Pages are really just two values of the same underlying `post_type` column, nothing stops a developer from defining a third. This chapter delivers on that promise directly, plus its exact counterpart for Categories and Tags.

register_post_type() — Defining a Third Content Category

```php
<?php
function register_portfolio_post_type() {
    register_post_type( 'portfolio', array(
        'labels' => array(
            'name'          => 'Portfolio Items',
            'singular_name' => 'Portfolio Item',
        ),
        'public'      => true,
        'has_archive' => true,
        'supports'    => array( 'title', 'editor', 'thumbnail' ),
        'menu_icon'   => 'dashicons-portfolio',
    ) );
}
add_action( 'init', 'register_portfolio_post_type' );
?>
```

THIS IS THE EXACT REVEAL WORDPRESS FUNDAMENTALS 3 PREVIEWED

Once this runs, `portfolio` becomes a real, valid value for the very same `post_type` column already holding `post` and `page`. A Portfolio Item isn't stored in some separate, new system — it's an ordinary row in the exact same database table, distinguished the same way Posts and Pages always have been.

- **public** — whether this post type is queryable and has its own front-end pages at all
- **has_archive** — whether an automatic archive listing page exists for it
- **supports** — which built-in editor features this post type gets (title, the main content editor, a featured image, and more)

NOTICE THE HOOK

`register_post_type()` is called inside a function hooked to WordPress's own `init` action — the same actions/filters mechanism WordPress Fundamentals 6 previewed as a black box. WordPress Intermediate/Advanced 5 formally covers this mechanism in full; for now, just recognize the pattern — WordPress runs your function at the right moment, rather than your code calling WordPress directly at the top level of a file.

What This Does to the Template Hierarchy

Registering `portfolio` makes `single-portfolio.php` and `archive-portfolio.php` real, meaningful filenames in WordPress Intermediate/Advanced 2's own hierarchy — the exact same lookup system already covered, now simply extended with one more post type to check against.

register_taxonomy() — Custom Classification, Beyond Categories and Tags

A **taxonomy** is a classification system — and per WordPress Fundamentals 7, Categories and Tags were always just two pre-built taxonomies with different configurations. This chapter's own function lets you build your own.

```
<?php
function register_project_type_taxonomy() {
    register_taxonomy( 'project_type', 'portfolio', array(
        'labels'       => array( 'name' => 'Project Types' ),
        'hierarchical' => true,
        'public'       => true,
    ) );
}
add_action( 'init', 'register_project_type_taxonomy' );
?>
```

HIERARCHICAL SETTING	BEHAVES LIKE
`true`	Categories, per WordPress Fundamentals 7 — parent/child structure allowed
`false`	Tags, per WordPress Fundamentals 7 — flat, no hierarchy

CATEGORIES AND TAGS WERE NEVER SPECIAL

"Category" and "Tag" are simply the names of two taxonomies WordPress core happens to register automatically, using exactly this same `register_taxonomy()` mechanism. Setting `hierarchical` to `true` or `false` on a custom taxonomy is, mechanically, the same choice that separates Categories from Tags in the first place.

Bringing It Together

A `portfolio` custom post type combined with a `project_type` custom taxonomy gives a real, working content model: individual portfolio items, each classified into a hierarchical set of project types — the same pattern behind products, testimonials, team members, or events on a real WordPress site.

AN HONEST ALTERNATIVE WORTH KNOWING

Plugins like Custom Post Type UI exist to configure exactly this — post types and taxonomies — entirely through the dashboard, with no PHP at all, echoing WordPress Fundamentals 6's own plugin material. Writing it by hand, as this chapter does, gives full control and keeps the definition in version-controlled code rather than database settings — a real trade-off, not a right-or-wrong choice.

Hands-On Exercises

EXERCISE 1

Write the register_post_type() call for a "Testimonial" custom post type that supports only a title and the main editor, with no automatic archive page. Explain each argument you chose.

View solution

EXERCISE 2

Explain why registering a "portfolio" post type doesn't require creating any new database table, connecting your answer directly to WordPress Fundamentals 3's own material.

View solution

EXERCISE 3

A developer registers a custom taxonomy called "skill_level" with hierarchical set to false. Explain what this taxonomy will behave like, using this chapter's own comparison to Categories and Tags.

View solution

CHAPTER 4 QUICK REFERENCE

- **register_post_type()**, hooked to init — defines a new value for the same post_type column Posts and Pages already share
- public, has_archive, and supports control queryability, archive pages, and which editor features the new post type gets
- A registered post type gets its own real, meaningful entries in the template hierarchy — single-{type}.php, archive-{type}.php

- **register_taxonomy()** — Categories and Tags were always just two pre-built taxonomies; hierarchical true/false is the exact setting separating them
- Custom Post Type UI exists as a no-code, plugin-based alternative to writing this by hand — a real trade-off, not a wrong approach
- **Next chapter:** Plugin Development Fundamentals

CHAPTER 5 OF 10

Plugin Development Fundamentals

WordPress Intermediate/Advanced

Chapter 5 · Plugin Development Fundamentals

`add_action( 'init', ... )` has appeared twice already, unexplained, in WordPress Intermediate/Advanced 1 and 4. This is the chapter that formally opens it up — actions and filters, the single most important WordPress-specific concept a developer needs, and the mechanism behind every plugin WordPress Fundamentals 6 ever installed as a finished product.

What a Plugin Actually Is, Structurally

Just like a theme's `style.css` from WordPress Intermediate/Advanced 1, a plugin needs its own specially-formatted comment header — placed at the top of a PHP file inside `wp-content/plugins/` — that WordPress parses to recognize and register it.

```
<?php
/**
 * Plugin Name: My First Plugin
 * Description: A minimal example plugin.
 * Version: 1.0
 * Author: Your Name
 */
```

Everything below that header is ordinary PHP — and a plugin does its actual work almost entirely by hooking functions onto actions and filters, rather than running code directly at the top level of the file.

Actions vs. Filters — The Core Distinction

ACTIONS	FILTERS
Let you **do** something at a specific point	Let you **modify** a piece of data as it passes through
The hooked function's return value is ignored	The hooked function **must return a value** — WordPress uses whatever comes back
Registered with `add_action()`	Registered with `add_filter()`

ACTIONS	FILTERS
Example: sending an email when a post is published	Example: shortening every post excerpt to 20 words

add_action() — Doing Something

Every prior use of `add_action( 'init', ... )` in this course follows the same shape: `add_action( 'hook_name', 'function_to_run' )`. WordPress calls your function at the exact moment it reaches that named point in its own execution.

```
<?php
function add_footer_credit() {
    echo '<p>Built with a custom plugin.</p>';
}
add_action( 'wp_footer', 'add_footer_credit' );
```

This directly reuses `wp_footer()` from WordPress Intermediate/Advanced 1 — that function's entire job is firing the `wp_footer` action, giving every plugin hooked into it a chance to run.

add_filter() — Modifying Data

```
<?php
function custom_excerpt_length( $length ) {
    return 20;
}
add_filter( 'excerpt_length', 'custom_excerpt_length' );
```

WordPress calls this function while generating an excerpt, passing in the current word-count limit; the function returns `20`, and that's the value WordPress actually uses.

> **THE SINGLE MOST COMMON FILTER MISTAKE**
>
> A filter function that forgets to `return` a value doesn't leave the data unchanged — it silently replaces it with `null`, since a PHP function with no explicit return implicitly returns nothing. A filter hooked onto `the_content` that echoes debugging output instead of returning the (possibly modified) content will wipe out every post's body text entirely, with no error message anywhere pointing back to the cause.

A Small, Real Plugin — Both Mechanisms Together

```php
<?php
/**
 * Plugin Name: Reading Time Estimator
 * Description: Prepends an estimated reading time to every post.
 * Version: 1.0
 */

// A filter: modifies the post content
function prepend_reading_time( $content ) {
    if ( is_single() ) {
        $word_count = str_word_count( strip_tags( $content ) );
        $minutes    = ceil( $word_count / 200 );
        $content    = "<p>Estimated reading time: {$minutes} min</p>" . $content;
    }
    return $content;
}
add_filter( 'the_content', 'prepend_reading_time' );

// An action: logs when the plugin loads, does nothing visible
function log_plugin_loaded() {
    error_log( 'Reading Time Estimator plugin loaded.' );
}
add_action( 'init', 'log_plugin_loaded' );
```

This is genuinely the same category of tool WordPress Fundamentals 6 covered installing — the only difference is that this one was built from scratch, using nothing beyond ordinary PHP and the two hook functions this chapter just covered.

Priority — What Happens When Multiple Functions Hook the Same Point

`add_action()` and `add_filter()` both accept an optional priority argument (default `10`) — lower numbers run earlier. Multiple plugins commonly hook the same action or filter, and priority is how their relative order is controlled when it matters.

```php
add_action( 'wp_footer', 'add_footer_credit', 20 ); // runs later than the default
```

Hands-On Exercises

EXERCISE 1

A developer writes a filter function hooked to the_title that's supposed to append " — Read More" to every post title, but forgets to include a return statement. Explain exactly what visitors would see as a result.

View solution

EXERCISE 2

Explain, using this chapter's own compare table, whether sending a notification email when a comment is posted should be built as an action or a filter, and why.

View solution

EXERCISE 3

Explain what add_action('init', 'register_portfolio_post_type') from WordPress Intermediate/Advanced 4 is actually doing, now that this chapter has formally explained the mechanism behind it.

View solution

CHAPTER 5 QUICK REFERENCE

- A plugin file needs its own comment header block, parsed by WordPress, exactly like a theme's style.css
- **Actions** (add_action()) — do something at a specific point; return value ignored
- **Filters** (add_filter()) — modify a value as it passes through; the hooked function must return a value
- A filter that forgets to return silently wipes the data to null — the single most common filter mistake
- Priority (default 10, lower runs earlier) controls execution order when multiple functions hook the same point
- Every register_post_type()/register_taxonomy() call from Chapter 4 was always this exact mechanism, hooked to init
- **Next chapter:** The WordPress REST API

CHAPTER 6 OF 10

The WordPress REST API

WordPress Intermediate/Advanced

Chapter 6 · The WordPress REST API

Every prior chapter has covered WordPress generating HTML through PHP templates. This chapter covers its other, built-in way of exposing content: as structured JSON, through a real, genuine REST API — no plugin required, and a perfect concrete application of everything *API Types & Design* already covered in the abstract.

It's Already There, On Every WordPress Site

Since WordPress 4.7 (2016), every WordPress install automatically exposes its own REST API at `/wp-json/` — nothing needs to be installed or enabled for basic read access to work.

ENDPOINT	RETURNS
`/wp-json/wp/v2/posts`	A list of published posts, as JSON
`/wp-json/wp/v2/posts/42`	A single post by ID
`/wp-json/wp/v2/pages`	A list of pages
`/wp-json/wp/v2/categories`	A list of categories
`/wp-json/wp/v2/users`	A list of users (public fields only, by default)

Genuine REST, Not a WordPress-Specific Invention

This is a real, concrete instance of everything *API Types & Design* already covered: resources identified by URLs, HTTP methods carrying meaning (`GET` to read, `POST` to create, `PUT` / `PATCH` to update, `DELETE` to remove), and JSON as the response format.

```
{
  "id": 42,
  "date": "2024-03-15T10:30:00",
  "title": { "rendered": "Hello World" },
  "content": { "rendered": "<p>Post body as HTML</p>" },
  "excerpt": { "rendered": "<p>Post body as HTML&hellip;</p>" },
  "author": 1,
  "featured_media": 17,
```

```
  "categories": [ 3, 5 ]
}
```

Authentication — Reading Is Open, Writing Isn't

`GET` requests to public content work with no authentication at all — anyone can fetch the JSON above directly. Creating, editing, or deleting content requires real authentication: application passwords (a WordPress-generated credential meant specifically for API access) or a cookie-plus-nonce combination for requests made from a logged-in browser session.

Exposing a Custom Post Type — A Real, Easy-to-Miss Gotcha

SHOW_IN_REST DEFAULTS TO FALSE

The `portfolio` custom post type from WordPress Intermediate/Advanced 4 does *not* automatically appear in the REST API — `register_post_type()`'s own `show_in_rest` argument defaults to `false`. Forgetting this argument is a genuine, common source of confusion: the post type works perfectly in the dashboard and on the front end, yet `/wp-json/wp/v2/portfolio` simply doesn't exist until it's explicitly enabled.

```
register_post_type( 'portfolio', array(
    // ...all the arguments from Chapter 4...
    'show_in_rest' => true,
) );
```

A First Look at Headless WordPress

Because content is available as structured JSON, a completely separate front-end application — built in React, Vue, or any other framework, with no PHP templates and no theme system involved at all — can consume that JSON directly instead of relying on WordPress's own HTML rendering. This pattern is called **headless WordPress**: WordPress used purely as a content-management backend, with the actual presentation layer built entirely separately.

SCOPE NOTE

This course stops at recognizing the pattern and knowing the REST API is what makes it possible. Actually building a headless front-end is a genuinely large, separate topic — a real application of general front-end framework skills against this exact API, not something this WordPress-focused course goes on to build.

Hands-On Exercises

EXERCISE 1

A developer registers a "portfolio" custom post type exactly as shown in WordPress Intermediate/Advanced 4, then visits /wp-json/wp/v2/portfolio expecting to see JSON data, but gets an error instead. Explain why, and what's missing.

View solution

EXERCISE 2

Explain why a GET request to /wp-json/wp/v2/posts works with no authentication, while a POST request to the same endpoint requires it, connecting your answer to what each HTTP method actually represents.

View solution

EXERCISE 3

Explain what "headless WordPress" actually means, and specifically what role the REST API plays in making it possible at all.

View solution

CHAPTER 6 QUICK REFERENCE

- Every WordPress site exposes a real REST API at /wp-json/ automatically, since WordPress 4.7
- Genuine REST: resources as URLs, HTTP methods carry meaning, JSON responses — a real application of API Types & Design's own material
- GET requests to public content need no authentication; write operations require application passwords or cookie+nonce
- show_in_rest defaults to false — a custom post type needs it explicitly set to true to appear in the REST API at all
- **Headless WordPress** — a separate front-end framework consuming the REST API directly, with no PHP templates or theme involved
- **Next chapter:** Enqueuing Scripts & Styles Properly

CHAPTER 7 OF 10

Enqueuing Scripts & Styles Properly

WordPress Intermediate/Advanced

Chapter 7 · Enqueuing Scripts & Styles Properly

WordPress Intermediate/Advanced 1's `header.php` called `wp_head()` so plugins could inject their own output — but that hook alone doesn't stop a theme from loading its own CSS and JavaScript the wrong way. This chapter covers the correct mechanism, and the real conflicts it exists specifically to prevent.

The Wrong Way — Hardcoding Tags Directly

```
<!-- inside header.php — don't do this -->
<link rel="stylesheet" href="/wp-content/themes/my-theme/style.css">
<script src="/wp-content/themes/my-theme/js/main.js"></script>
```

This works, in the narrow sense that the browser will load the files — but it has real, concrete problems: no dependency management, no way to prevent the same library being loaded twice by a theme and a plugin that both need it, and no way for WordPress or anything else to know these files were loaded at all.

The Right Way — wp_enqueue_script() / wp_enqueue_style()

```
<?php
function my_theme_scripts() {
    wp_enqueue_style(
        'my-theme-style',
        get_stylesheet_uri(),
        array(),
        '1.0'
    );

    wp_enqueue_script(
        'my-theme-script',
        get_template_directory_uri() . '/js/main.js',
        array( 'jquery' ),
        '1.0',
        true
    );
}
```

```
add_action( 'wp_enqueue_scripts', 'my_theme_scripts' );
?>
```

Note the hook — `wp_enqueue_scripts`, a dedicated action specifically for this purpose, distinct from the `init` hook Chapters 4 and 5 already used for post types and general setup.

PARAMETER	WHAT IT DOES
Handle	A unique name identifying this specific file — how WordPress and other plugins refer to it
Source URL	Where the file actually lives — get_stylesheet_uri()/get_template_directory_uri() reference the active theme's own files correctly, rather than a hardcoded path
Dependencies	An array of handles that must load first — the actual mechanism preventing the wrong-way problems above
Version	Appended to the URL for cache-busting — bump it when the file changes so browsers don't serve a stale cached copy
In footer (scripts only)	Loading JavaScript near the end of the page rather than the head — directly relevant to *Performance & Core Web Vitals*'s own material

Dependencies — Where the Real Conflict Prevention Happens

JQUERY IS ALREADY REGISTERED — USE IT, DON'T RELOAD IT

WordPress core ships with jQuery already bundled and registered under the handle `jquery`. Listing `array( 'jquery' )` as a script's dependency does two things at once: it guarantees jQuery loads before your script regardless of enqueue order, and — critically — if jQuery is already enqueued by anything else (another plugin, the theme itself), WordPress won't load a second copy. This is the concrete, code-level fix for exactly the kind of resource-duplication conflict WordPress Fundamentals 6 only ever described as a symptom to troubleshoot.

WHAT THE HARDCODED VERSION ACTUALLY RISKS

Two plugins each hardcoding their own `<script src="jquery.js">` tag, with no shared handle and no dependency system involved, can genuinely load two separate copies of jQuery on the same page — wasted bandwidth at best, and real, hard-to-diagnose JavaScript conflicts at worst, since two independently-loaded copies of the same library don't share state with each other.

Referencing Theme Files Correctly

`get_stylesheet_uri()` returns the active theme's own `style.css` URL, and `get_template_directory_uri()` returns the theme folder's own base URL — both resolve correctly regardless of what the site's actual domain

or folder structure happens to be, unlike a hardcoded path that would break the moment the site moved or the folder structure changed.

Hands-On Exercises

EXERCISE 1

A theme hardcodes its own jQuery-dependent script tag directly in header.php, and a separately installed plugin also enqueues jQuery properly via wp_enqueue_script(). Explain what happens on the page as a result, using this chapter's own material.

View solution

EXERCISE 2

Explain why a developer would bump a script's version parameter from '1.0' to '1.1' after editing the file, and what would go wrong for returning visitors if they forgot to.

View solution

EXERCISE 3

Explain why wp_enqueue_scripts, not init, is the correct hook for enqueuing scripts and styles, based on what this chapter says about dedicated hooks.

View solution

CHAPTER 7 QUICK REFERENCE

- Never hardcode <link>/<script> tags directly — no dependency management, real risk of duplicate-library conflicts
- wp_enqueue_style()/wp_enqueue_script(), hooked to wp_enqueue_scripts (not init)
- Handle, source URL, dependencies array, version (cache-busting), in_footer (scripts) — the five real parameters
- jQuery ships pre-registered under the handle 'jquery' — depending on it avoids duplicate loads across theme and plugins
- get_stylesheet_uri()/get_template_directory_uri() reference theme files correctly, unlike a hardcoded path
- **Next chapter:** Security Hardening

CHAPTER 8 OF 10

Security Hardening

WordPress Intermediate/Advanced

Chapter 8 · Security Hardening

WordPress Fundamentals 8 described roles as "named bundles of capabilities" and promised the real mechanism would come later, in real code. This chapter is that payoff — plus three more mechanisms giving WordPress-specific teeth to security principles this site has already taught in general: request forgery protection, output escaping, and safe database queries.

Capability Checks — current_user_can()

```php
<?php
if ( current_user_can( 'publish_posts' ) ) {
    // show the publish button
}

if ( current_user_can( 'manage_options' ) ) {
    // show admin-only settings
}
?>
```

This is the exact function WordPress Fundamentals 8 promised: real code checking a specific capability, not a role name. "Editor" was never checked directly anywhere in WordPress core — every permission decision comes down to a call exactly like this one.

HIDING A BUTTON IS NOT THE SAME AS SECURING AN ACTION

Hiding an "Edit" link from the dashboard for a user without the right capability is a usability nicety, not a security boundary. The actual privileged action itself — the code that would edit or delete something — must independently check `current_user_can()` before doing anything, regardless of what was or wasn't shown in the interface. A user who reaches the underlying action URL directly, bypassing the hidden button entirely, must still be blocked by the check itself.

Nonces — WordPress's Own CSRF Protection

A **nonce** ("number used once") is a token WordPress generates and verifies to confirm a request genuinely originated from the expected page and user — this is WordPress's own concrete implementation of exactly the defense *CSRF — Cross-Site Request Forgery* covers in general.

```
<!-- generating, inside a form -->
<?php wp_nonce_field( 'my_action_name', 'my_nonce_field' ); ?>
```

```
<?php
// verifying, on submission
if ( ! wp_verify_nonce( $_POST['my_nonce_field'], 'my_action_name' ) ) {
    wp_die( 'Security check failed.' );
}
?>
```

COMBINE BOTH CHECKS TOGETHER

A genuinely secure privileged action checks *both* the nonce (proving the request came from the right place) *and* `current_user_can()` (proving the requester is actually allowed to do this) — neither one alone is a complete defense.

Output Escaping — WordPress's Own Answer to XSS

`esc_html()`, `esc_attr()`, and `esc_url()` are concrete, CMS-specific instances of exactly the context-dependent output encoding *XSS — Cross Site Scripting* already covers in general — different contexts need different escaping.

```
<?php
echo '<h1>' . esc_html( $title ) . '</h1>';
echo '<a href="' . esc_url( $link ) . '">Click here</a>';
echo '<input value="' . esc_attr( $value ) . '">';
?>
```

FUNCTION	CONTEXT
`esc_html()`	Plain text inside an HTML element's body
`esc_attr()`	Text placed inside an HTML attribute
`esc_url()`	A URL, most often inside an `href` or `src` attribute

THE WRONG ESCAPING FUNCTION IS STILL A REAL VULNERABILITY

Using `esc_html()` on data being placed inside an attribute (rather than `esc_attr()`) doesn't provide the correct protection for that specific context — matching exactly the "encode for the context, not generically" lesson *XSS — Cross Site Scripting* already taught in the abstract.

$wpdb->prepare() — WordPress's Own Parameterized Queries

`$wpdb->prepare()` is WordPress's own direct implementation of the parameterized-query defense *SQL Injections and Defences* covers in general — never concatenate user input directly into a SQL string, in WordPress exactly as anywhere else.

```
<?php
global $wpdb;

$results = $wpdb->get_results(
    $wpdb->prepare(
        "SELECT * FROM {$wpdb->prefix}posts WHERE post_author = %d",
        $user_id
    )
);
?>
```

`%d`, `%s`, and `%f` are placeholders for integers, strings, and floats respectively — `$wpdb` safely substitutes the actual value in, exactly the same underlying protection as any other language's parameterized query API.

One Pattern, Four Mechanisms

THE SYNTHESIS THIS CHAPTER HAS BEEN BUILDING TOWARD

Every mechanism in this chapter is the same underlying move: a general security principle already taught elsewhere on this site, given a specific, concrete WordPress implementation. Capability checks are least-privilege access control, per *Database Security*'s own material. Nonces are CSRF protection. Output escaping is context-aware XSS defense. `$wpdb->prepare()` is SQL injection prevention. None of these are WordPress inventing new security ideas — they're WordPress giving already-general principles a real, callable API, echoing *OWASP Top 10*'s own framing and directly connecting back to the hardening discipline *Setting Up a Web Server on Debian* applied at the server layer.

Hands-On Exercises

EXERCISE 1

A developer hides a "Delete Post" button from users without the right role, but the underlying delete action itself never calls current_user_can(). Explain the real, exploitable gap this leaves.

View solution

EXERCISE 2

A developer uses esc_html() to output a URL inside an href attribute instead of esc_url(). Explain why this is still a real vulnerability, connecting your answer to this chapter's own context-dependent escaping material.

View solution

EXERCISE 3

Explain why this chapter describes $wpdb->prepare() as "WordPress giving an already-general principle a real, callable API" rather than a uniquely WordPress security concept.

View solution

CHAPTER 8 QUICK REFERENCE

- **current_user_can()** — the real mechanism behind roles, delivering on WordPress Fundamentals 8's own promise; hiding UI is not a substitute for checking it directly in the privileged action
- **Nonces** (wp_nonce_field(), wp_verify_nonce()) — WordPress's own CSRF protection; combine with capability checks, never rely on either alone
- **esc_html()/esc_attr()/esc_url()** — context-dependent output escaping, WordPress's own answer to XSS
- **$wpdb->prepare()** — WordPress's own parameterized queries, WordPress's own answer to SQL injection
- All four are WordPress-specific implementations of general principles already taught in this site's own security courses, not new inventions
- **Next chapter:** Performance & Caching

CHAPTER 9 OF 10

Performance & Caching

WordPress Intermediate/Advanced

Chapter 9 · Performance & Caching

WordPress Fundamentals 6 named caching plugins as an essential category without explaining why they matter so much specifically for WordPress. This chapter explains exactly that — and maps every technique directly onto *Performance & Core Web Vitals*'s own LCP, INP, and CLS metrics, rather than treating "performance" as one vague, undifferentiated goal.

Why WordPress Specifically Has a Performance Problem

Per WordPress Fundamentals 1's own LAMP-stack material, every WordPress page load, by default, involves real PHP execution and real MySQL queries — WordPress rebuilds the page dynamically on every single request unless something intervenes. A static HTML file requires none of this; a default WordPress page genuinely does, every time, for every visitor.

Page Caching — The Single Biggest Lever

Page caching stores the fully-rendered HTML output of a page after its first generation, then serves that static copy directly for subsequent requests — skipping PHP execution and MySQL queries entirely for every visitor after the first.

DIRECTLY TARGETS LCP

Server response time is a direct, measured contributor to Largest Contentful Paint — the slower the server takes to generate a page, the later the largest visible element can possibly appear. Skipping PHP/MySQL execution via page caching is one of the most reliable, highest-impact ways to improve LCP on a WordPress site specifically.

THE CLASSIC HARD PROBLEM: CACHE INVALIDATION

A cached page has to be regenerated or cleared the moment its underlying content actually changes — a new comment, an edited post. A caching setup that doesn't correctly invalidate stale pages will confidently serve outdated content indefinitely, with no visible error anywhere.

Object Caching — A Different, Complementary Layer

Page caching mostly benefits anonymous, logged-out visitors — logged-in users often see personalized dashboard content that can't be cached wholesale the same way. **Object caching** solves a related but different problem: caching the results of individual, expensive database queries in memory, so the same query run repeatedly across many requests doesn't have to hit MySQL every single time.

> **PERSISTENT VS. NON-PERSISTENT**
>
> WordPress ships with a basic, built-in object cache — but it's non-persistent, meaning it only lasts for the duration of a single page request and provides no benefit across separate requests. A genuinely persistent object cache (commonly backed by Redis or Memcached) is what actually delivers repeated-query savings across many different visitors and requests over time.

Image Optimization — Also Mostly an LCP Story

The largest visible element on a page is very often an image — a hero image, a featured post image — making image handling directly relevant to LCP as well.

- **Lazy loading** — deferring images below the initial viewport until they're actually needed, via the native `loading="lazy"` attribute
- **Responsive images** — serving an appropriately-sized image per device, using the `srcset` attribute WordPress automatically generates from the multiple image sizes WordPress Fundamentals 7 already covered
- **Compression** — reducing file size without materially harming visual quality, often handled by a dedicated plugin

CLS — A WordPress-Specific Risk Worth Naming

> **MISSING WIDTH/HEIGHT ATTRIBUTES**
>
> WordPress automatically outputs explicit width and height attributes on images in most standard contexts — a real, built-in CLS protection, since the browser can reserve the correct space before the image finishes loading. A common mistake in custom theme development is stripping these attributes out (often unintentionally, through custom image-output code), reintroducing the exact layout-shift risk WordPress's own defaults were already preventing.

INP — A Lighter Touch, Tying Back to Chapter 7

Unnecessary or poorly-optimized JavaScript execution is the primary driver of poor INP. WordPress Intermediate/Advanced 7's own proper enqueuing material — loading scripts in the footer, only enqueuing what a specific page actually needs — is directly relevant here; INP isn't a separate new problem so much as a real-world payoff of getting Chapter 7's own material right.

Mapping Techniques to Metrics

TECHNIQUE	PRIMARILY HELPS
Page caching	LCP (server response time)
Object caching	LCP, indirectly, for dynamic/logged-in content page caching can't cover
Lazy loading, responsive images, compression	LCP
Preserving width/height on images	CLS
Proper script enqueuing (Chapter 7)	INP

Hands-On Exercises

EXERCISE 1

A site enables page caching, but visitors keep seeing an old version of a post for hours after it was edited. Explain what's most likely wrong, using this chapter's own material.

View solution

EXERCISE 2

Explain why page caching alone doesn't solve performance for logged-in users, and what a persistent object cache adds that WordPress's own built-in object cache doesn't.

View solution

EXERCISE 3

A custom theme's own image-output code strips out the width and height attributes WordPress normally adds automatically. Explain which specific Core Web Vitals metric this most directly harms, and why.

View solution

CHAPTER 9 QUICK REFERENCE

- WordPress rebuilds pages dynamically via PHP/MySQL on every request by default, unlike a static file
- **Page caching** — the single biggest LCP lever, skipping PHP/MySQL entirely on repeat requests; cache invalidation is the classic hard problem

- **Object caching** — caches individual query results, benefits logged-in/dynamic content page caching can't cover; needs a persistent backend (Redis/Memcached) to help across requests
- Lazy loading, responsive srcset images, and compression are all primarily LCP techniques
- Missing width/height attributes on images is a real, WordPress-specific CLS risk
- Proper script enqueuing (Chapter 7) is directly what improves INP
- **Next chapter:** Capstone — Building a Custom Theme With a Custom Post Type

CHAPTER 10 OF 10

Capstone: Building a Custom Theme With a Custom Post Type

WordPress Intermediate/Advanced

Chapter 10 · Capstone — Building a Custom Theme With a Custom Post Type

Nine chapters have built the pieces, one at a time. This capstone assembles them into one real, working theme — a portfolio site with a custom post type, a widget area, and a genuinely secured contact form — closing both this course and the full 20-chapter WordPress track.

1. Theme File Structure

```
/*
Theme Name: Capstone Portfolio
Author: Your Name
Version: 1.0
Text Domain: capstone-portfolio
*/
```

`style.css`'s header, `index.php`, and `header.php` / `footer.php` (correctly calling `wp_head()` / `wp_footer()`) follow exactly the structure Chapter 1 established.

2. The "Project" Custom Post Type & Taxonomy

```
<?php
function register_project_post_type() {
    register_post_type( 'project', array(
        'labels'       => array( 'name' => 'Projects' ),
        'public'       => true,
        'has_archive'  => true,
        'supports'     => array( 'title', 'editor', 'thumbnail' ),
        'show_in_rest' => true,
    ) );

    register_taxonomy( 'project_type', 'project', array(
        'labels'       => array( 'name' => 'Project Types' ),
        'hierarchical' => true,
        'public'       => true,
```

```
    ) );
}
add_action( 'init', 'register_project_post_type' );
?>
```

Directly from Chapter 4, with Chapter 6's `show_in_rest` included from the start — this project type is REST-accessible immediately, not bolted on later.

3. Displaying Projects — Template Hierarchy & The Loop

```
<!-- archive-project.php -->
<?php get_header(); ?>

<?php if ( have_posts() ) : ?>
  <?php while ( have_posts() ) : the_post(); ?>
    <h2><a href="<?php the_permalink(); ?>"><?php the_title(); ?></a></h2>
    <?php the_post_thumbnail( 'medium' ); // preserves width/height automatically ?>
  <?php endwhile; ?>
<?php endif; ?>

<?php get_footer(); ?>
```

Per Chapter 2's own hierarchy, this file is automatically selected for the projects archive with no further wiring required. `the_post_thumbnail()` deliberately preserves the width/height attributes Chapter 9 named as a real CLS protection.

4. A Widget Area

```
<?php
function register_sidebar_area() {
    register_sidebar( array(
        'name' => 'Portfolio Sidebar',
        'id'   => 'portfolio-sidebar',
    ) );
}
add_action( 'widgets_init', 'register_sidebar_area' );
?>
```

`register_sidebar()`, hooked to its own dedicated `widgets_init` action, defines the widget area WordPress Fundamentals 7 covered purely from the site-owner side — this is what actually makes a widget area exist for a theme to offer in the first place.

5. A Properly Secured Contact Form

```
<!-- the form, inside a template -->
<form method="post">
  <?php wp_nonce_field( 'contact_form_submit', 'contact_nonce' ); ?>
  <input type="text" name="contact_name">
  <textarea name="contact_message"></textarea>
  <button type="submit">Send</button>
</form>
```

```
<?php
function handle_contact_form() {
    if ( ! isset( $_POST['contact_nonce'] ) ||
         ! wp_verify_nonce( $_POST['contact_nonce'], 'contact_form_submit' ) ) {
        wp_die( 'Security check failed.' );
    }

    $name    = sanitize_text_field( $_POST['contact_name'] );
    $message = sanitize_textarea_field( $_POST['contact_message'] );

    wp_mail(
        get_option( 'admin_email' ),
        'New contact form submission',
        esc_html( $name ) . ' wrote: ' . esc_html( $message )
    );
}
add_action( 'admin_post_nopriv_contact_form', 'handle_contact_form' );
add_action( 'admin_post_contact_form', 'handle_contact_form' );
?>
```

Chapter 8's own two-part defense, fully assembled: the nonce is checked before anything else runs, and both the submitted values and the final email output are escaped. This is the single most security-sensitive piece of the entire capstone, and deliberately the one built with the most care.

6. Enqueuing the Theme's Own Assets

```
<?php
function portfolio_theme_scripts() {
    wp_enqueue_style( 'portfolio-style', get_stylesheet_uri(), array(), '1.0' );
    wp_enqueue_script(
        'portfolio-script',
        get_template_directory_uri() . '/js/main.js',
        array( 'jquery' ),
        '1.0',
        true // in_footer — Chapter 9's own INP-relevant choice
```

```
    );
}
add_action( 'wp_enqueue_scripts', 'portfolio_theme_scripts' );
?>
```

Chapter Attribution

PIECE	SOURCE CHAPTER
Theme file structure, wp_head()/wp_footer()	Chapter 1
archive-project.php selected automatically	Chapter 2
The Loop displaying projects	Chapter 3
The "project" post type and "project_type" taxonomy	Chapter 4
Contact form handled via a hooked action	Chapter 5
show_in_rest on the project post type	Chapter 6
Properly enqueued styles/scripts with a jQuery dependency	Chapter 7
Nonce verification and output escaping on the contact form	Chapter 8
Preserved thumbnail dimensions, footer-loaded scripts	Chapter 9

Honest Scope Note

WHAT THIS CAPSTONE DELIBERATELY DOESN'T COVER

- No WooCommerce/e-commerce — reserved for a still-outstanding, deliberately-not-yet-scoped third WordPress course
- No Full Site Editing/block-theme implementation — this capstone builds a classic PHP-template theme, matching this course's own focus throughout
- No automated testing or version-controlled deployment workflow
- Production deployment itself is *Setting Up a Web Server on Debian*'s own territory, not repeated here

A CLOSING, HONEST NOTE ON REAL-WORLD PRACTICE

A genuine production version of a theme like this would often be built as a child theme of an established base theme, per WordPress Fundamentals 5's own material, rather than entirely from scratch — this capstone builds from scratch deliberately, specifically to demonstrate every piece explicitly rather than to model the most efficient real-world starting point.

Hands-On Exercises

EXERCISE 1

Explain why the contact form's nonce check happens before the sanitize_text_field() calls, rather than after, referencing this chapter's own Chapter 8 material.

View solution

EXERCISE 2

Explain why the_post_thumbnail() is specifically called out in this chapter as preserving width/height "automatically," and why that matters for this capstone's own Chapter 9 attribution.

View solution

EXERCISE 3

Explain why this capstone's own closing note says a real production theme would likely be a child theme instead, and why the capstone deliberately doesn't build it that way.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE & TRACK COMPLETE

- A real theme combining theme anatomy, the template hierarchy, the Loop, a custom post type + taxonomy, the REST API, proper enqueuing, security hardening, and performance practice
- The contact form is the single most security-sensitive piece — nonce check first, then sanitize input, then escape output
- register_sidebar(), hooked to widgets_init, is what makes a widget area exist for a theme to offer
- WooCommerce/e-commerce is deliberately out of scope — a plausible future third WordPress course, not yet planned in detail
- This completes both WordPress Fundamentals and WordPress Intermediate/Advanced — the full 20-chapter track