BEYOND ZERO SERIES

PHP Beyond Zero

AN INTERMEDIATE GUIDE TO

MODERN PHP AND WEB APPLICATION DEVELOPMENT

LANCASTER SOLUTIONS LLC

PHP

BEYOND ZERO

Intermediate PHP Application Architecture, Security, Data, and Delivery

Expanded Professional Edition

Architecture • maintainability • testing • security • performance • deployment • guided labs • production capstone

Curtis Wayne Lancaster II

Published by Lancaster Solutions LLC

Beyond Zero Series • Practical technology education for real-world builders

About This Expanded Edition

> **AUDIENCE:** Readers who completed PHP From Zero or can build forms, sessions, PDO CRUD, classes, Composer projects, and basic tests.

This revised edition treats PHP as a production application platform, connecting language design to HTTP middleware, databases, authorization, queues, observability, framework tradeoffs, and secure deployment.

Technical baseline

Modern PHP 8.4 and 8.5 practices, with framework-independent architecture and current official runtime guidance.

What “intermediate” means in this series

You can already write small working programs or pages. This book focuses on what happens when the work must be changed safely, tested repeatedly, deployed predictably, and understood by someone other than its original author.

The objective is not to memorize every feature. It is to build judgment: choosing boundaries, stating contracts, designing failure behavior, gathering evidence, and balancing complexity against the real needs of a project.

Examples are intentionally compact. The labs, projects, and capstone ask you to supply missing production details such as validation, logging, authorization, migration, accessibility, or rollback. Those omissions are part of the learning design rather than permission to ignore them.

How to use the book

- Read the chapter thesis first and write down what you expect the technique to improve.
- Run or reproduce the example in a small, disposable project before transplanting it into a larger codebase.
- Complete the guided lab with tests and notes. Then attempt the independent challenge without copying the lab structure mechanically.
- Keep an engineering notebook containing assumptions, decisions, failed experiments, measurements, and follow-up work.
- Treat the capstone as a portfolio-quality delivery: source code is only one deliverable alongside evidence, documentation, and an operational plan.

Publication and educational use

Contents and Learning Roadmap

The roadmap is organized into four stages. Each stage introduces decisions that the next stage assumes you can explain and verify.

Stage	Focus	Coverage	Core topics
Part 1	Structure, Models, and Contracts	Chapters 1–4	Composer, PSR Conventions, and Application Boundaries, Strict Types, Value Objects, Enums, and Readonly Design, Composition, Interfaces, and SOLID Without Ceremony, HTTP Kernels, Middleware, Routing, and Request Lifecycles
Part 2	Boundaries, Data, and Collaboration	Chapters 5–8	Framework Choices: Laravel, Symfony, and Deliberate Adoption, PDO, Transactions, Repositories, and Query Design, ORMs, Mapping, and Aggregate Persistence, Authentication, Sessions, Authorization, and Tenancy
Part 3	Reliability, Scale, and Verification	Chapters 9–12	Forms, CSRF, Uploads, and Context-Specific Output Safety, JSON APIs, Serialization, and Contract Evolution, Queues, Events, Scheduling, and Reliable Background Work, Caching, Performance, and PHP Runtime Behavior
Part 4	Delivery, Operations, and Evolution	Chapters 13–16	Testing, Static Analysis, and Mutation Thinking, Configuration, Logging, Error Handling, and Observability, Deployment, Containers, Migrations, and Operational Readiness, Architecture Decisions and Sustainable PHP Systems

PART IV — Delivery, Operations, and Evolution

Projects, capstone, and references

PART I

Structure, Models, and Contracts

Chapters 1–4

CHAPTER 1
Composer, PSR Conventions, and Application Boundaries

> **CHAPTER THESIS:** Intermediate PHP begins when autoloading, package metadata, namespaces, coding standards, and layer direction are reproducible rather than maintained through manual require statements.

Learning outcomes

- Explain how composer metadata shapes boundaries, testability, and failure behavior in a maintainable implementation.
- Apply PSR-4 autoloading in a focused implementation and identify the contract it creates for callers.
- Compare an implementation that uses layer direction with a simpler alternative, then justify the added complexity.
- Evaluate the operational and maintenance consequences of standards as coordination under realistic change and failure.
- Evaluate the composer Project example, identify assumptions about composer metadata and PSR-4 autoloading, and justify one viable alternative.
- For “Composer, PSR Conventions, and Application Boundaries”, produce evidence using PHPUnit results, static-analysis output, migration checks, HTTP contract examples, and deployment smoke tests; state what the evidence proves and what remains uncertain.

The intermediate shift

The intermediate challenge in “Composer, PSR Conventions, and Application Boundaries” is not learning one more API; it is deciding where the behavior belongs and what must remain true when the surrounding system changes.

Composer metadata, PSR-4 autoloading, Layer direction, and Standards as coordination reinforce one another, but they optimize different risks. The chapter asks you to decide which concern owns each rule, where translation occurs, and how the design behaves when one assumption changes.

Credible evidence for “Composer, PSR Conventions, and Application Boundaries” includes PHPUnit results, static-analysis output, migration checks, HTTP contract examples, and deployment smoke tests. Capture the setup and expected result so another reader can reproduce the claim instead of trusting a successful demonstration.

Composer metadata

composer.JSON records package identity, runtime requirements, autoload mapping, dependencies, scripts, and development tools.

Implementation guidance. Declare the PHP and extension versions actually supported, commit composer.lock for applications, and test installation with production dependencies only.

For Composer metadata, the design payoff is controlled change: callers depend on the contract while the implementation can evolve behind it.

PSR-4 autoloading

PSR-4 maps namespace prefixes to directories so class loading follows a predictable convention.

Implementation guidance. Match case exactly, keep one primary class per file, and avoid adding files with side effects to the autoload section.

Operationally, PSR-4 autoloading must define failure ownership, retained context, retry safety, and the signal shown to users or operators.

Layer direction

Domain and application code should not depend on controllers, templates, frameworks, or PDO implementations.

Implementation guidance. Place interfaces near their consumers and implement them in infrastructure, then wire concrete services in a bootstrap file or container.

Verify Layer direction through its narrowest public contract, then add adversarial cases that exercise malformed input and dependency failure.

Standards as coordination

PSR recommendations help libraries and teams interoperate, but they do not replace application design.

Implementation guidance. Adopt formatting, logging, HTTP message, and container conventions when they improve compatibility and tool support.

For maintenance, Standards as coordination should communicate ownership through names, types, modules, and documentation rather than institutional memory.

Worked pattern

COMPOSER PROJECT

```
{
  "name": "lancaster/study-portal",
  "type": "project",
  "require": {
    "php": "^8.4 || ^8.5"
  },
  "autoload": {
    "psr-4": { "Study\\": "src/" }
  },
  "autoload-dev": {
    "psr-4": { "Study\\Tests\\": "tests/" }
  }
}
```

Walkthrough

1. The production and test namespaces are separated.
2. The runtime constraint states supported PHP branches.
3. Composer generates the autoloader; application code includes vendor/autoload.PHP once.

Design decisions to notice

- Composer metadata: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- PSR-4 autoloading: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Layer direction: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Standards as coordination: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- In the composer Project, which decisions express the policy described in “Composer, PSR Conventions, and Application Boundaries”, and which are replaceable mechanisms behind the boundary?
- For “Composer, PSR Conventions, and Application Boundaries”, which guarantees are supplied by PHP, Composer, and modern web frameworks, and which still require explicit validation, cleanup, monitoring, or documentation?

Failure modes and diagnostic thinking

- Using classmap to hide a disorganized namespace layout.
- Committing vendor while also expecting Composer installation.
- A domain namespace importing framework request objects.
- Running Composer as a privileged production user with unnecessary plugins enabled.

DIAGNOSTIC EXERCISE: Reproduce this risk in the smallest useful example: Using classmap to hide a disorganized namespace layout. Identify the first observable symptom, the earliest detection boundary, one preventive control, and one operational signal that would distinguish recurrence from a different failure.

Guided lab

> **OBJECTIVE:** Convert a manually required project to PSR-4 namespaces and a clean Composer install.

Phase 1: Clarify the contract

Turn the objective into acceptance criteria: Convert a manually required project to PSR-4 namespaces and a clean Composer install. Name trusted and untrusted inputs, explicit non-goals, and the result a reviewer can observe.

Phase 2: Create a baseline

Capture a baseline for “Composer, PSR Conventions, and Application Boundaries” using PHPUnit results, static-analysis output, migration checks, HTTP contract examples, and deployment smoke tests. Preserve the command, fixture, environment, or screenshot needed to repeat it.

Phase 3: Implement the smallest safe change

Implement the smallest coherent change that advances the objective. Keep composer metadata behind a named boundary and verify each increment before adding the next.

Phase 4: Verify normal and hostile conditions

Exercise the normal case plus malformed input, boundary values, and a realistic failure involving PSR-4 autoloading. Include cancellation, timeout, rollback, fallback, or recovery where the chapter makes it relevant.

Phase 5: Review and communicate

Review the evidence with the objective in view. Record the chosen design, the rejected alternative, remaining risk, and the signal that would justify revisiting layer direction.

Independent challenge

Add scripts for test, static analysis, style checks, and a production install smoke test.

> **CONSTRAINT:** Complete the independent challenge with a different framework adapter from the worked pattern. Explain how the change affects failure behavior, testing evidence, and maintenance cost.

Stretch challenge

Create architecture checks preventing Domain from depending on Infrastructure or Framework namespaces.

Chapter review

1. What problem does composer metadata solve, and what new complexity can it introduce when overused?
2. What problem does PSR-4 autoloading solve, and what new complexity can it introduce when overused?
3. What problem does layer direction solve, and what new complexity can it introduce when overused?
4. What problem does standards as coordination solve, and what new complexity can it introduce when overused?
5. Which evidence would demonstrate that composer metadata remains correct under failure, and which part still requires representative measurement or human review?
6. Under what project constraints would a simpler alternative to standards as coordination be the more responsible choice?

Definition of done

- The objective is demonstrated for the normal path and failure cases relevant to composer metadata.
- The contract around PSR-4 autoloading is explicit, smaller than its implementation, and exercised by evidence.
- The verification does not depend on hidden secrets, machine-specific paths, or uncontrolled state while testing layer direction.
- A reviewer can trace the decision about standards as coordination from requirement to implementation and evidence.
- Known limitations, operational signals, and follow-up work for “Composer, PSR Conventions, and Application Boundaries” are recorded with an owner or review trigger.