src/Domain/Money.php

```php
final readonly class Money
{
    public function __construct(
        public int $cents,
        public string $currency = 'USD',
    ) {}

    public function add(Money $o): self
    { return new self($this->cents + $o->cents); }
}

// strict types · enums · PDO · PSR-15
```

REST API

Web Store

Domain · PDO

Tests · Docker

```
$ composer require monolog/monolog
$ vendor/bin/phpunit OK (all tests passed)
$ php -S localhost:8000 -t public
```

A PRACTICAL GUIDE TO PHP 8

Modern PHP from Zero to Production

Learn modern PHP and build, test, and ship a real marketplace web application

PHP 8.4 · Strict Types · Enums · Composer · PDO
REST API · PHPUnit · PHPStan · Redis · Docker

LANGUAGE · WEB · DATA · PRODUCTION

THENEXBOOKS

Modern PHP from Zero to Production

A Practical Guide to PHP 8 for Building Real Web Applications
From your first script to a tested, secured, containerized marketplace — built step by step

First Edition

THENEXBOOKS

Modern PHP from Zero to Production
A Practical Guide to PHP 8 for Building Real Web Applications

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. PHP, its extensions, Composer, frameworks, and third-party packages discussed in this book evolve; always consult the official documentation for the versions you use.

PHP is a project of The PHP Group; this book is independent and is not affiliated with, sponsored by, or endorsed by The PHP Group, the PHP Foundation, PHP-FIG, or any framework or package maintainer. Laravel is a trademark of Taylor Otwell. Symfony is a trademark of Symfony SAS. MySQL is a trademark of Oracle Corporation. MariaDB is a trademark of MariaDB plc. Redis is a trademark of Redis Ltd. Docker is a trademark of Docker, Inc. NGINX is a trademark of F5, Inc. GitHub is a trademark of GitHub, Inc. Microsoft, Visual Studio Code, and Windows are trademarks of the Microsoft group of companies. JetBrains and PhpStorm are trademarks of JetBrains s.r.o. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Orbit Store, the online marketplace used as the running example in this book, is fictional, as are its sellers, products, customers, orders, and the "OrbitPay" payment provider. Email addresses use reserved example domains.

Preface

PHP has been quietly running a large share of the web for decades. Blogs and news sites, online shops, learning platforms, internal business tools, and APIs behind mobile apps — a great deal of it is served by PHP. The language is easy to start with, cheap to host, and fast to iterate on, which is exactly why so many projects begin with it and keep growing on it.

PHP also carries an old reputation: inconsistent functions, spaghetti code mixed with HTML, SQL injection waiting to happen. Some of that came from the early days, and some from tutorials that never moved on. The language itself has changed enormously. Modern PHP has strict typing, enums, readonly properties and classes, attributes, `match` expressions, property hooks, a mature package manager in Composer, shared standards from PHP-FIG, excellent static analysis, and frameworks that rival anything in other ecosystems. Written the modern way, PHP is clean, safe, and genuinely pleasant to work with.

This book teaches PHP the modern way from the very first chapter. We never start with the old habits and then ask you to unlearn them. Instead, we build one real application step by step, introducing each idea at the moment it solves a problem we actually have, and explaining *why* it works the way it does.

Who This Book Is For

- **Beginners with a little programming or HTML experience** who want to build real, dynamic websites and web APIs.
- **Developers from other languages** such as JavaScript, Python, Java, or C# who need to work on PHP code or want a productive server-side language.
- **Developers who learned "old PHP"** and want to catch up: strict types, Composer, namespaces, enums, readonly classes, PDO, PHP 8.4 property hooks, and modern tooling.
- **Students and career changers** who want a portfolio-quality project built the way professional teams build software — tested, secured, and deployed.

What You Will Learn

- How PHP runs: the CLI, the built-in development server, PHP-FPM, and the request lifecycle.
- The language from the ground up: types and strict typing, strings, arrays, control flow, functions, and closures.
- Object-oriented PHP: classes, readonly value objects, interfaces, traits, enums, namespaces, Composer, and exceptions.
- Modern PHP 8 features: `match`, named arguments, attributes, first-class callables, property hooks, and asymmetric visibility.

- Web fundamentals without magic: requests and responses, a front controller and router, safe templates, forms, file uploads, sessions, and authentication.
- Data and persistence: PDO and prepared statements, schema design and migrations, repositories, transactions, caching with APCu and Redis, JSON, dates, and money.
- Real-world applications: a REST API, dependency injection, PSR standards and middleware, background jobs, email, payment webhooks, and how Laravel and Symfony map onto it all.
- Production PHP: PHPUnit testing, static analysis, coding standards, debugging, logging, security hardening, OPcache and performance, Docker, Nginx with PHP-FPM, and CI/CD.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up PHP and covers the language fundamentals. **Part II — Object-Oriented PHP** (Chapters 6–10) covers classes, interfaces, traits, enums, namespaces and Composer, exceptions, and modern PHP 8 features. **Part III — Building for the Web** (Chapters 11–15) covers HTTP, routing, templates, forms, file uploads, sessions, and authentication. **Part IV — Data and Persistence** (Chapters 16–20) covers PDO, schema design, repositories and transactions, caching, JSON, dates, and money. **Part V — Real-World Applications** (Chapters 21–24) covers a REST API, dependency injection and PSR middleware, background jobs and webhooks, and the major frameworks. **Part VI — Quality and Production** (Chapters 25–28) covers testing, static analysis and debugging, security and performance, and deployment with a complete case study. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we build **Orbit Store**, a fictional online marketplace where independent sellers — Northwind Ceramics, Pixel Forge Prints, and Cedar & Twine — list their products, and customers can buy from several sellers in a single order. It starts as a handful of command-line scripts working with plain arrays. Over the chapters it grows into a clean object-oriented domain model, a server-rendered web store with a cart, login, and checkout, a MySQL-backed application with transactions that never oversell stock, a REST API with token authentication and rate limiting, and a system that sends emails, processes payment webhooks, and calculates seller payouts in background jobs. In Part VI we test it, analyze it, harden it, and ship it in containers behind Nginx and PHP-FPM.

Versions and Conventions

Examples target **PHP 8.4**, and we always say when a feature needs PHP 8.1 or newer, so you can follow along on PHP 8.3 with only small adjustments. We use **Composer 2**, **MySQL 8** or **MariaDB** (with **SQLite** for quick local work and tests), and recent versions of PHPUnit and PHPStan. Tools and frameworks release often, so where details may change between versions, a Note points you to the official documentation rather than guessing.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `src/Domain/Money.php`, tell you where the code lives. Terminal commands work the same on Windows, macOS, and Linux unless noted.

Contents

Preface

PART I — GETTING STARTED

1 Why PHP? The Language, Its Ecosystem, and Your Setup

2 Your First Scripts: The CLI, the Built-in Server, and How PHP Runs

3 Types, Variables, Operators, and Strict Typing

4 Strings, Arrays, and Working with Data

5 Control Flow, Functions, and Closures

PART II — OBJECT-ORIENTED PHP

6 Classes, Objects, Constructors, and Readonly Properties

7 Interfaces, Abstract Classes, Traits, and Enums

8 Namespaces, Autoloading, and Composer

9 Errors, Exceptions, and Defensive Code

10 Modern PHP 8 Features: match, Attributes, Property Hooks, and More

PART III — BUILDING FOR THE WEB

11 HTTP in PHP: Requests, Responses, and Superglobals

12 Routing and the Front Controller

13 Templates and Views: Escaping, Layouts, and Twig

14 Forms, Validation, and File Uploads

15 Sessions, Cookies, and Authentication

PART IV — DATA AND PERSISTENCE

16 Databases with PDO: Connections, Queries, and Prepared Statements

17 Designing the Orbit Store Schema and Running Migrations

18 Repositories, Transactions, and the Checkout Flow

19 Caching with APCu and Redis

20 JSON, Dates and Time Zones, and Money Done Right

PART V — REAL-WORLD APPLICATIONS

21 Building a REST API for Orbit Store

22 Dependency Injection, PSR Standards, and Middleware

23 Background Jobs, Email, and Payment Webhooks

24 Frameworks in Practice: Laravel and Symfony

PART VI — QUALITY AND PRODUCTION

25 Testing with PHPUnit (and Pest)

26 Static Analysis, Coding Standards, Debugging, and Logging

27 Security Hardening and Performance Tuning

28 Deployment: Docker, Nginx + PHP-FPM, CI/CD, and the Final Case Study

APPENDICES

A PHP Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Getting Started

What PHP is good at and how to set up a modern PHP environment, how PHP runs on the command line and behind a web server, and the fundamentals every PHP program is built from: types and strict typing, strings, arrays, control flow, functions, and closures.

1 · Why PHP?

2 · Your First Scripts

3 · Types, Variables, and Strict Typing

4 · Strings, Arrays, and Data

5 · Control Flow, Functions, and Closures

CHAPTER 1 · PART I — GETTING STARTED

Why PHP? The Language, Its Ecosystem, and Your Setup

Before writing a single line of code, it pays to understand what PHP is, why it still runs so much of the web, and how the language you will learn differs from the PHP of fifteen years ago. By the end of this chapter you will have a working PHP 8.4 installation, Composer, a configured editor, and a clear picture of Orbit Store, the marketplace we will build together across the whole book.

What PHP Is

PHP is a general-purpose scripting language that grew up on the web server. Its original job was simple: take a request from a browser, run a small program, and send back HTML. That job has never really changed, but the language around it has. Today PHP is used to build server-rendered websites, JSON APIs, command-line tools, background workers, and the back ends of some of the most widely deployed content management systems in existence.

The name started life as "Personal Home Page" tools, written in the mid-1990s as a handful of helpers for tracking visits to a personal website. As the project grew into a real language, the name was turned into the recursive acronym "PHP: Hypertext Preprocessor". You will still see the original "preprocessor" model in Chapter 2, when we mix PHP and HTML in one file, but most modern PHP code looks much more like the code you would write in Java, C#, TypeScript, or Python: classes, interfaces, type declarations, packages, and tests.

PHP is **interpreted** in the sense that you do not run a separate compile step: you save a file and run it. Under the hood, the Zend Engine compiles your source into internal *opcodes* and executes them. The OPcache extension keeps those opcodes in shared memory so the next request does not parse the file again; in production it is always on, and we tune it in Chapter 27. PHP 8 also includes a JIT compiler, which matters less for typical web applications that mostly wait on databases and networks.

Shared-nothing: one request, one clean slate

The single most important idea in PHP's design is the **shared-nothing request lifecycle**. In the classic setup, every HTTP request starts with a fresh, empty PHP environment. Your script runs from the top, builds whatever objects it needs, produces a response, and then everything is thrown away: variables, objects, open files, database connections (unless you opt into persistent connections). The next request starts again from nothing.

Diagram: the classic PHP request lifecycle

```
Browser               Web server              PHP worker
-------               ----------              ----------
GET /products  ---->  Nginx / Apache   ---->  start fresh
                                               load code (OPcache)
                                               run index.php
                                               query database
                                               build HTML
200 OK + HTML  <----  pass response    <----  send output
                                               free everything
                                               (ready for next request)
```

This model has real advantages. A bug that corrupts some state in one request cannot leak into the next one. Memory leaks are mostly harmless, because memory is released at the end of each request. Scaling out is straightforward: add more PHP worker processes, or more servers, and put a load balancer in front. There is no shared in-memory state to synchronize, so anything that must survive between requests goes into a database, a cache such as Redis, a session store, or a file.

The model also has a cost: every request must bootstrap the application again, loading configuration and connecting to the database. Autoloading and OPcache make this fast enough for most applications. Long-running runtimes such as FrankenPHP, RoadRunner, and Swoole or OpenSwoole keep your application in memory between requests instead; they are legitimate options with their own trade-offs, but this book is written for the classic model first, because that is where you should start.

What PHP Is Good At

Every language has a sweet spot. PHP's sweet spot is the request-response web application, and it has been tuned for that job for three decades. Here is what that means in practice:

- **Web applications and APIs.** Reading a request, talking to a database, and returning HTML or JSON is exactly what PHP was designed for. The standard library includes functions for URLs, headers, cookies, sessions, JSON, dates, and hashing out of the box.
- **Content management systems.** WordPress, Drupal, Joomla, and MediaWiki (the software behind Wikipedia) are all written in PHP. If you work in web development long enough, you will almost certainly touch one of them.
- **Fast iteration.** Save the file, refresh the browser. There is no build step for the server code, which keeps the feedback loop short, especially while learning.
- **Cheap and ubiquitous hosting.** Almost every hosting company supports PHP, from inexpensive shared hosting to managed platforms and container services. Deployment can be as simple as copying files or as sophisticated as a container pipeline (Chapter 28).
- **A mature ecosystem.** Composer, the dependency manager, and Packagist, its public package registry, give you access to a very large number of well-maintained libraries. Frameworks such as Laravel and Symfony provide complete, battle-tested toolkits for building serious applications.

It is fair to say PHP powers a large share of the web, through those content management systems and through custom applications built on Laravel, Symfony, and other frameworks. You will not find usage percentages in this book, because those figures change constantly. PHP is less natural for heavy numeric computing, machine learning, or desktop GUIs, where other languages are usually a better fit.

From PHP 5 to PHP 8: The Modernization Story

If you wrote PHP many years ago, or you have only seen PHP in old tutorials, you may remember a language full of surprises: functions with inconsistent names, silent type conversions, the `mysql_*` functions, and code that mixed SQL, HTML, and business logic in a single file. Much of that reputation was earned. Much of it is also out of date.

The language has changed dramatically, step by step:

Era	What changed
PHP 5.x	A real object model, exceptions, PDO for databases, and later namespaces, closures, and traits.
PHP 6	Never released. An ambitious Unicode rewrite was abandoned, and the next version was numbered 7.
PHP 7.0	A rewritten engine with large speed and memory improvements, scalar type declarations, return types, `strict_types`, `??`, and `<=>`. The old `mysql_*` functions were removed.
PHP 7.1 to 7.4	Nullable types, `void`, typed properties, arrow functions, and `??=`.
PHP 8.0	Union types, named arguments, `match`, nullsafe `?->`, constructor promotion, attributes, the JIT, and saner string-to-number comparisons.
PHP 8.1	Enums, `readonly` properties, first-class callables, `never`, and fibers.
PHP 8.2	Readonly classes, DNF types, and the deprecation of dynamic properties.
PHP 8.3	Typed class constants, `#[\Override]`, and `json_validate()`.
PHP 8.4	Property hooks, asymmetric visibility, `array_find()` and friends, and a new HTML5 parser.

The overall direction is clear: PHP has become stricter, faster, and more expressive. Modern PHP code has types on every parameter and return value, uses classes and interfaces to model its domain, loads code through Composer's autoloader, talks to databases through prepared statements, and is checked by static analysis tools and automated tests. That is the PHP this book teaches.

This book targets **PHP 8.4** as its baseline. Almost everything also works on PHP 8.3; whenever we use a feature added in PHP 8.1 or later, we will say which version it needs. PHP 8.5 was released in late 2025 and adds the pipe operator `|>`, among other things. We will show it briefly in Chapter 10, clearly labeled, but none of the main code depends on it, so you can follow along on 8.4 without worrying.

> **NOTE**
>
> If you are coming from "old PHP", the biggest mental shift is not syntax. It is the toolchain. Composer replaces copying library files around, PSR-4 autoloading replaces long lists of `require` statements, PDO prepared statements replace string-built SQL, and tools like PHPUnit and PHPStan catch bugs before your users do. We introduce each of these at the point where you need it.

The release cycle

PHP ships a new minor version roughly once a year, usually toward the end of the year. Each version receives a period of active support, during which bugs and security issues are fixed, followed by a period of security-only fixes. After that, the version reaches end of life and no longer receives any updates, not even for security problems. The exact durations have been adjusted over time, so rather than memorizing them, check the "Supported Versions" page on php.net whenever you plan an upgrade or a new project.

The practical lesson is simple: **do not run an unsupported PHP version in production**. Upgrading once a year is much easier than jumping several versions at once, and tools such as Rector (Chapter 10 and Chapter 26) can automate much of the work.

Installing PHP

You need PHP 8.4 on your machine to follow along. There are several good ways to get it, and the right one depends on your operating system and taste. Pick one; you do not need all of them. Whichever you choose, the goal is the same: typing `php -v` in a terminal prints a version that starts with 8.4 (or 8.3 if you must).

Windows

On Windows you have three common options.

1. **Official zip builds.** The windows.php.net download page offers zip archives of PHP for Windows. For command-line use and the built-in development server, choose the 64-bit *Non Thread Safe* (NTS) build. Extract it to a folder such as `C:\php`, add that folder to your `PATH` environment variable, and open a new terminal.
2. **All-in-one development environments.** Tools such as Laragon and XAMPP bundle PHP with a web server and a database. They are convenient if you want everything in one installer. Make sure the bundled PHP version is 8.4 or 8.3, and that the `php` command you type in the terminal is the one you expect.
3. **WSL (Windows Subsystem for Linux).** If you prefer a Linux environment, install a distribution such as Ubuntu through WSL and follow the Linux instructions below inside it. Many PHP developers on Windows work this way, because production servers usually run Linux.

With the zip build, PHP ships with two sample configuration files, `php.ini-development` and `php.ini-production`. Copy `php.ini-development` to `php.ini` in the same folder, then open it and enable the extensions this book uses by removing the leading semicolon from the relevant lines:

C:\php\php.ini (excerpt)

```
extension_dir = "ext"

extension=curl
extension=intl
extension=mbstring
extension=openssl
extension=pdo_mysql
extension=pdo_sqlite
extension=sodium

date.timezone = UTC
```

macOS

The most common route on macOS is Homebrew. Once Homebrew is installed, a single command installs the current PHP release along with the most commonly needed extensions:

Terminal

```
$ brew install php
$ php -v
```

Homebrew tracks the latest PHP release. If the current release is newer than 8.4 that is fine for this book; if you need a specific older line, Homebrew and third-party taps offer versioned formulas, and tools that manage multiple PHP versions exist as well.

Linux

Every major Linux distribution packages PHP. On Debian and Ubuntu, the command-line interpreter and the extensions we need are separate packages:

Terminal

```
$ sudo apt update
$ sudo apt install php-cli php-mbstring php-xml php-intl php-curl \
      php-sqlite3 php-mysql
$ php -v
```

The catch is that a distribution's default PHP version is fixed when that distribution is released, so a long-term-support release may ship an older PHP than you need. On Ubuntu, many developers use the widely known PPA maintained by Ondřej Surý to get newer PHP versions; on other distributions,

similar repositories exist. Check the documentation of whichever repository you choose, since package names include the version (for example, packages along the lines of `php8.4-cli`). On Fedora and related distributions, `sudo dnf install php-cli` is the starting point.

Docker

If you already use Docker, you can skip installing PHP entirely and use the official image. The `php:8.4-cli` image contains the command-line interpreter:

Terminal

```
$ docker run --rm php:8.4-cli php -v
$ docker run --rm -it -v "$PWD":/app -w /app php:8.4-cli php hello.php
```

The second command mounts your current folder into the container at `/app` and runs `hello.php` from there. Docker is a great way to try several PHP versions side by side and is how we will deploy Orbit Store in Chapter 28, though a native installation usually feels quicker for day-to-day learning.

> **TIP**
> If you have several PHP versions installed, the one that runs is whichever `php` your shell finds first on the `PATH`. On macOS and Linux, `which php` shows its location; on Windows, `where php` does the same. When something behaves strangely, checking which binary you are actually running is a good first step.

Checking Your Installation

Three commands tell you almost everything about a PHP installation. Run them now and keep the output in mind; you will come back to them whenever something does not work.

`php -v`: the version

Terminal

```
$ php -v
```

You should see something like the following. The exact patch number, build date, and compiler details will differ on your machine:

Output (illustrative)

```
PHP 8.4.x (cli) (built: ...) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.4.x, Copyright (c) Zend Technologies
    with Zend OPcache v8.4.x, Copyright (c), by Zend Technologies
```

The `(cli)` part matters. PHP comes in several flavors called *SAPIs* (Server APIs): the CLI for the terminal, FPM for serving web requests behind Nginx or Apache, and others. Each SAPI can load a different configuration file, which is a common source of confusion: an extension may be enabled for the CLI but not for the web server, or the other way around.

`php -m`: the loaded extensions

PHP's functionality is split into extensions. Some, such as the core string and array functions, are always present. Others, such as `intl` or `pdo_mysql`, must be installed and enabled. `php -m` lists the modules that are currently loaded:

Terminal

```
$ php -m
[PHP Modules]
Core
ctype
curl
date
...
mbstring
openssl
PDO
pdo_mysql
pdo_sqlite
...

[Zend Modules]
Zend OPcache
```

Your list will be longer and may be ordered differently. For this book, make sure you see at least `mbstring`, `intl`, `PDO`, `pdo_sqlite`, `openssl`, `curl`, and `sodium`. You will need `pdo_mysql` from Chapter 16 onward if you want to use MySQL or MariaDB rather than SQLite. Note that JSON support has been part of PHP's core since PHP 8.0, so there is no separate extension to enable.

`php --ini`: the configuration files

PHP reads its settings from a file called `php.ini`, plus optional extra `.ini` files in a scan directory. `php --ini` tells you exactly which files the CLI loaded:

Terminal

```
$ php --ini
Configuration File (php.ini) Path: /etc/php/8.4/cli
Loaded Configuration File:         /etc/php/8.4/cli/php.ini
Scan for additional .ini files in: /etc/php/8.4/cli/conf.d
Additional .ini files parsed:      /etc/php/8.4/cli/conf.d/10-opcache.ini,
...
```

The paths above are typical of a Debian-style Linux system; yours will look different on Windows or macOS. If the "Loaded Configuration File" line says `(none)`, PHP is running with built-in defaults, which usually means you have not yet created a `php.ini` file. Two more commands are handy: `php -i` prints the full configuration (pipe it through `grep` or `findstr` to search it), and `php -r` runs a snippet of code directly from the command line.

Terminal

```
$ php -r 'echo PHP_VERSION, PHP_EOL;'
8.4.x
$ php -i | grep display_errors
display_errors => STDOUT => STDOUT
```

Development settings in `php.ini`

While learning and developing, you want PHP to be loud about problems. Make sure your development `php.ini` contains the following settings (the `php.ini-development` template already sets most of them):

php.ini (development)

```
error_reporting = E_ALL
display_errors = On
display_startup_errors = On
log_errors = On
memory_limit = 256M
date.timezone = UTC
```

`error_reporting = E_ALL` asks PHP to report every kind of error, warning, notice, and deprecation. `display_errors = On` prints them on screen so you see them immediately. In production you will do the opposite: hide errors from visitors and write them to a log instead (Chapter 27). Setting `date.timezone` to UTC avoids warnings and surprises when working with dates, and matches how Orbit Store stores timestamps.

> **WARNING**
>
> Never copy a development `php.ini` with `display_errors = On` to a production server. Error messages can reveal file paths, SQL queries, and other internals to anyone who visits your site. Production servers should start from `php.ini-production`, which keeps errors out of the response and in the log.

A First Sanity Check Script

Let's write our first PHP file. It is not "Hello, World" yet (that comes in Chapter 2), but a small script that checks whether your environment is ready for the rest of the book. Create a folder called `orbit-store` anywhere you like, and inside it a file called `check-env.php`:

orbit-store/check-env.php

```
<?php

declare(strict_types=1);

$required = ['mbstring', 'intl', 'pdo_sqlite', 'openssl', 'curl', 'sodium'];
$optional = ['pdo_mysql', 'opcache', 'xdebug'];

echo 'PHP version: ' . PHP_VERSION . PHP_EOL;

if (version_compare(PHP_VERSION, '8.4.0', '>=')) {
    echo 'OK: PHP 8.4 or newer' . PHP_EOL;
} elseif (version_compare(PHP_VERSION, '8.3.0', '>=')) {
    echo 'OK with notes: PHP 8.3 (a few 8.4 features will not work)' . PHP_EOL;
} else {
    echo 'Please upgrade: this book needs PHP 8.3 or newer' . PHP_EOL;
}

foreach ($required as $extension) {
    $status = extension_loaded($extension) ? 'loaded' : 'MISSING';
    echo str_pad($extension, 12) . $status . PHP_EOL;
}

foreach ($optional as $extension) {
    $status = extension_loaded($extension) ? 'loaded' : 'not loaded (optional)';
    echo str_pad($extension, 12) . $status . PHP_EOL;
}

echo 'php.ini: ' . (php_ini_loaded_file() ?: '(none)') . PHP_EOL;
```

Run it from inside the `orbit-store` folder:

Terminal

```
$ php check-env.php
PHP version: 8.4.x
OK: PHP 8.4 or newer
mbstring    loaded
intl        loaded
pdo_sqlite  loaded
openssl     loaded
curl        loaded
sodium      loaded
pdo_mysql   loaded
opcache     loaded
xdebug      not loaded (optional)
php.ini: /etc/php/8.4/cli/php.ini
```

Your output will differ in the details. Do not worry about understanding every line yet; Chapters 2 through 5 explain variables, arrays, `if` statements, `foreach` loops, and functions. A few things are worth noticing already. The file starts with `<?php`, which tells PHP where the code begins. The second line, `declare(strict_types=1);`, switches on strict type checking for this file; you will see it at the top

of every PHP file in this book, and Chapter 3 explains exactly what it does. `PHP_VERSION` and `PHP_EOL` are built-in constants: the running version and the platform's line ending. `extension_loaded()` asks whether an extension is available, and `php_ini_loaded_file()` returns the path of the loaded `php.ini`, or `false` when there is none, which is why we use the short ternary `?:` to fall back to `(none)`.

OPcache often shows as "not loaded" for the CLI, because it is usually enabled only for web requests; that is fine. If a *required* extension is missing, enable it in `php.ini` or install the matching package, then run the script again.

Installing Composer

Composer is PHP's dependency manager. It downloads the libraries your project needs, keeps track of their exact versions, and generates an autoloader so that your classes are loaded automatically. You will use it seriously from Chapter 8 onward, but install it now so your setup is complete.

- **Windows:** download and run `Composer-Setup.exe` from getcomposer.org. The installer finds your `php.exe` and adds Composer to your `PATH`.
- **macOS:** `brew install composer`.
- **Linux and others:** follow the command-line instructions on the getcomposer.org download page. They download an installer, verify its checksum, run it, and remove it. Because the checksum changes with each installer release, copy the commands from the site rather than from a book. Then move the resulting `composer.phar` to a folder on your `PATH` as `composer`. Some distributions also package Composer, though often an older release.

Check that it works:

Terminal

```
$ composer --version
Composer version 2.x.x ...
```

This book assumes Composer 2. If you see a version starting with 1, upgrade with `composer self-update` or reinstall it.

> **TIP**
>
> Avoid running Composer as the root or administrator user. Composer can execute scripts defined by packages, so it should run with the same ordinary permissions as the rest of your development work. Composer itself prints a warning if it detects that it is running as root.

Choosing an Editor

You can write PHP in any text editor, but a good editor with PHP support makes a huge difference while learning: it underlines mistakes as you type, completes function names, shows parameter types, and lets you jump to the definition of a class. Two choices are especially popular:

- **Visual Studio Code** is free and extensible. Install a PHP language extension; Intelephense is a widely used one, and there are others. Point the extension at your PHP executable if it asks, so it can match your installed version.
- **PhpStorm** is a commercial IDE dedicated to PHP, with deep refactoring support, built-in database tools, and debugger integration.

The choice is personal; this book does not depend on any particular editor. Whatever you choose, configure it to save files as UTF-8, use four spaces for indentation (the convention in the PHP-FIG coding style, PSR-12 and its successor PER Coding Style), and use Unix line endings (`\n`). In Chapter 26 we add automated formatters so you never argue about spaces again, and we set up step debugging with Xdebug.

Meet Orbit Store

Learning a language from disconnected snippets is like learning to cook by reading individual ingredients. In this book, almost every example contributes to a single project, so you can see how the pieces fit together and how a real codebase grows. That project is **Orbit Store**, a fictional online marketplace.

In Orbit Store, independent **sellers** list **products**, and **customers** can buy from several sellers in a single order. The marketplace takes a commission from each sale and pays the rest to the seller. This is a realistic, modestly complex domain: it has money, stock levels, discount coupons, shipping rules, order statuses, and multiple parties with different needs. It is big enough to justify good design, but small enough to understand completely.

The sellers

id	Name	Slug	Commission
1	Northwind Ceramics	northwind-ceramics	12%
2	Pixel Forge Prints	pixel-forge-prints	12%
3	Cedar & Twine	cedar-and-twine	10%

Notice the ampersand in "Cedar & Twine". That innocent character will teach us an important lesson about escaping output in Chapter 13.

The products

id	Seller	SKU	Name	Price	Stock
1	1	ORB-CER-001	Speckled Stoneware Mug	$24.00	35
2	1	ORB-CER-002	Glazed Serving Bowl	$48.00	12
3	2	ORB-PRT-001	Retro Rocket Poster	$19.99	80