PHP SERIES · COURSE 3

PHP Advanced

Composer in depth, design patterns, MVC from scratch, PHPUnit and mocking, security in depth, performance and caching, framework basics, REST APIs, and async/queues - capped with a tested REST API capstone. 10 complete chapters, each with coding challenges and solutions.

10 Chapters

osztromok.com

CHAPTER 1: COMPOSER AND PACKAGE MANAGEMENT IN DEPTH

COURSE 3 · CH 1

Composer and Package Management In Depth

Beyond autoloading — version constraints, composer.lock, scripts, and managing a real project's dependencies

Chapter 6 of Intermediate introduced Composer purely for PSR-4 autoloading. Composer's actual core purpose is **dependency management** — installing, updating, and constraining the versions of third-party packages a project relies on. This chapter covers that side properly.

composer require — Installing a Real Package

```
$ composer require monolog/monolog
# downloads the package into vendor/, adds it to composer.json,
# and updates composer.lock
```

This single command does three things: downloads `monolog/monolog` and its own dependencies into `vendor/`, records the package in `composer.json` under `"require"`, and writes the exact installed versions to `composer.lock`.

Semantic Versioning — How Package Versions Are Structured

Semantic versioning ("SemVer") gives every package version a predictable meaning, which is what makes version constraints in `composer.json` actually useful rather than arbitrary.

Version Constraints

CONSTRAINT	MEANING
`^2.4.1`	2.4.1 or higher, but below 3.0.0 (no breaking changes allowed)
`~2.4.1`	2.4.1 or higher, but below 2.5.0 (patch updates only)
`2.4.*`	Any 2.4.x version
`>=2.0`	Any version 2.0 or above, with no upper limit
`2.4.1`	That exact version only — rarely used, very restrictive

^ (CARET) IS THE STANDARD, RECOMMENDED DEFAULT

It allows genuinely useful updates (new features, bug fixes) to be installed automatically, while trusting SemVer's promise that a MAJOR version bump is the only kind of change allowed to break compatibility — and that kind of update is deliberately excluded.

composer.lock — Why It Matters, and Why It's Committed to Git

```
$ composer install
# if composer.lock exists, installs the EXACT versions it specifies
# (ignores composer.json's looser constraints entirely for this step)

$ composer update
# re-resolves composer.json's constraints to the newest allowed versions,
# and REWRITES composer.lock to match
```

UNLIKE VENDOR/, COMPOSER.LOCK SHOULD BE COMMITTED TO GIT — THE OPPOSITE RULE FROM VENDOR/

Without it, two developers running `composer install` on the same `composer.json` could end up with subtly different package versions (anything matching the constraints, but not necessarily identical) — a classic source of "works on my machine" bugs. Committing `composer.lock` guarantees everyone, including a production server, installs the exact same versions.

require vs require-dev — Separating Real Dependencies From Tooling

```
$ composer require --dev phpunit/phpunit
# installed under "require-dev" in composer.json — testing tools,
# not needed to actually RUN the application in production
```

```
// composer.json (excerpt)
{
  "require": {
    "monolog/monolog": "^2.4"
  },
  "require-dev": {
    "phpunit/phpunit": "^10.0"
  }
}
```

A production deployment typically runs `composer install --no-dev`, skipping everything in `require-dev` entirely — testing frameworks (covered properly in Chapter 4) and similar developer-only tools have no reason to be installed on a live server.

Composer Scripts — Shortcuts for Common Commands

```
{
  "scripts": {
    "test": "phpunit",
    "lint": "phpcs src/"
  }
}
```

```
$ composer test
# runs "phpunit" — a memorable, project-specific shortcut
```

The `"scripts"` section turns long or easily-forgotten commands into short, consistent ones that every contributor to a project can rely on, regardless of how the underlying tool is actually invoked.

Checking for Vulnerable Dependencies

```
$ composer audit
# checks installed packages against known security advisories
```

OUTDATED DEPENDENCIES ARE A GENUINE, COMMON SOURCE OF REAL VULNERABILITIES

A project can be written with perfect security practices and still be vulnerable purely because of an old version of a third-party library it depends on — running `composer audit` periodically (and especially before a deployment) is good practice, not paranoia.

Coding Challenges

CHALLENGE 1

For each of these version constraints, write out in your own words exactly which versions would be considered acceptable: ^1.2.0, ~1.2.0, 1.2.*. Then explain which one you'd choose for a new project dependency, and why.

View solution

CHALLENGE 2

Write a composer.json with both a "require" entry for a hypothetical package "acme/mailer" version ^3.0, and a "require-dev" entry for "phpunit/phpunit" version ^10.0. Add a "scripts" section with a "test" shortcut running phpunit. Explain in a comment what command would install only the production dependencies, skipping require-dev entirely.

View solution

CHALLENGE 3

Explain, in your own words and in detail, the practical difference between running "composer install" and "composer update" when a composer.lock file already exists — including which one a CI/CD pipeline or production deployment should normally use, and why getting this wrong could cause a "works on my machine" bug.

View solution

CHAPTER 1 QUICK REFERENCE

- **composer require pkg/name** — installs a package, updates composer.json and composer.lock
- **SemVer: MAJOR.MINOR.PATCH** — breaking / new feature / bug fix, by convention
- **^ (caret)** — recommended default constraint; allows non-breaking updates only
- **composer.lock** — commit this to git (unlike vendor/); guarantees identical installed versions everywhere
- **composer install** — installs exact locked versions; **composer update** — re-resolves and rewrites the lock file
- **require-dev / --no-dev** — separates dev-only tooling (tests, linters) from real production dependencies
- **"scripts" in composer.json** — memorable shortcuts for common project commands
- **composer audit** — checks installed packages against known security advisories
- **Next chapter:** design patterns in PHP — Singleton, Factory, Strategy, Repository

COURSE 3 · CH 2

Design Patterns in PHP: Singleton, Factory, Strategy, Repository

Named, reusable solutions to recurring object-design problems

A **design pattern** is a named, well-understood solution to a problem that comes up repeatedly in object-oriented code. Knowing the name and shape of a few common patterns means recognising the problem they solve — and being able to communicate a design decision to another developer in a single word, rather than re-explaining the whole structure every time.

Singleton	Factory	Strategy	Repository
Exactly one instance of a class, shared everywhere	Delegates object creation to a dedicated method/class	Swap an algorithm/behaviour at runtime via a common interface	Hides data-storage details behind a simple, collection-like API

Singleton — Exactly One Instance

```php
<?php
  class Config {
      private static ?Config $instance = null;
      private array $settings = [];

      private function __construct() {   // private — cannot be called with "new" from ou
          $this->settings = ['env' => 'production'];
      }

      public static function getInstance(): Config {
          if (self::$instance === null) {
              self::$instance = new self();
          }
          return self::$instance;
      }

      public function get(string $key) {
          return $this->settings[$key] ?? null;
```

```
        }
    }

    $config1 = Config::getInstance();
    $config2 = Config::getInstance();
    var_dump($config1 === $config2);   // true — genuinely the SAME object both times
?>
```

The constructor is `private`, so `new Config()` is impossible from outside the class — the only way to get an instance is `getInstance()`, which creates it once and returns that same object on every subsequent call.

SINGLETON IS GENUINELY CONTROVERSIAL — USE SPARINGLY

It introduces global, shared state, which can make code harder to test (since a test can't easily substitute a different `Config` without extra effort) and creates a hidden dependency between any two classes that both call `Config::getInstance()`. Dependency injection (passing the needed object in explicitly, rather than reaching for a global singleton) is usually the better-regarded modern approach — Singleton is shown here because it's a genuinely common pattern to recognise, not necessarily one to reach for by default.

Factory — Delegating Object Creation

```
<?php
    interface Notification {
        public function send(string $message);
    }

    class EmailNotification implements Notification {
        public function send(string $message) { echo "Email: $message<br>"; }
    }
    class SmsNotification implements Notification {
        public function send(string $message) { echo "SMS: $message<br>"; }
    }

    class NotificationFactory {
        public static function create(string $type): Notification {
            return match ($type) {
                'email' => new EmailNotification(),
                'sms'   => new SmsNotification(),
```

```
            default => throw new InvalidArgumentException("Unknown type: $type"),
        };
    }
}

$notifier = NotificationFactory::create('email');
$notifier->send("Your order has shipped");
?>
```

Calling code asks the factory for "an email notification" without needing to know — or directly reference — the specific `EmailNotification` class. This decouples the calling code from exactly which class gets instantiated, which becomes genuinely useful when that decision needs to vary (different config, different environment, a new notification type added later).

> **MATCH — A CLEANER ALTERNATIVE TO A LONG SWITCH FOR SIMPLE VALUE-TO-RESULT MAPPING**
>
> PHP's `match` expression (used above) is similar to `switch` from Fundamentals Chapter 3, but uses strict comparison by default, requires no `break`, and is itself an expression — it directly returns a value, rather than needing variables assigned inside each case.

Strategy — Swappable Behaviour Behind a Common Interface

```
<?php
interface DiscountStrategy {
    public function apply(float $price): float;
}

class PercentageDiscount implements DiscountStrategy {
    public function __construct(private float $percent) {}
    public function apply(float $price): float {
        return $price * (1 - $this->percent / 100);
    }
}

class FixedAmountDiscount implements DiscountStrategy {
    public function __construct(private float $amount) {}
    public function apply(float $price): float {
        return max(0, $price - $this->amount);
    }
}
```

```
    class ShoppingCart {
        public function __construct(private DiscountStrategy $discount) {}
        public function checkout(float $price): float {
            return $this->discount->apply($price);
        }
    }

    $cart = new ShoppingCart(new PercentageDiscount(10));
    echo $cart->checkout(100);   // 90
?>
```

`ShoppingCart` doesn't know or care which discount logic it's using — only that whatever it's given implements `DiscountStrategy` (Intermediate Chapter 2's interfaces, applied here). Swapping discount behaviour means passing a different object into the constructor; `ShoppingCart` itself never needs to change.

Repository — Hiding Storage Details Behind a Simple API

```
<?php
    interface UserRepository {
        public function find(int $id): ?array;
        public function save(array $user): void;
    }

    class MySqlUserRepository implements UserRepository {
        public function __construct(private PDO $pdo) {}
        public function find(int $id): ?array {
            $stmt = $this->pdo->prepare("SELECT * FROM users WHERE id = :id");
            $stmt->execute(['id' => $id]);
            return $stmt->fetch() ?: null;
        }
        public function save(array $user): void {
            // INSERT or UPDATE via PDO — full SQL omitted for brevity
        }
    }

    // Calling code only ever depends on the INTERFACE:
    function greetUser(UserRepository $repo, int $id) {
        $user = $repo->find($id);
        echo "Hello, " . ($user['name'] ?? 'stranger');
```

```
    }
?>
```

`greetUser()` works against the `UserRepository` interface — it has no idea whether data actually comes from MySQL, a JSON file, or an in-memory test array (a `FakeUserRepository` could be swapped in for testing). This is genuinely the same idea as Strategy, applied specifically to data access — and it's what makes automated testing of database-dependent code realistically feasible.

Coding Challenges

CHALLENGE 1

Implement a Singleton class called Logger with a private array property for storing log lines, a static getInstance() method, and a public method log(string $message) that appends to the array. Demonstrate that calling Logger::getInstance() from two different points in the script still shares the same log entries.

View solution

CHALLENGE 2

Create a Shape interface with a getArea() method, two implementations (Circle, Square), and a ShapeFactory with a static create(string $type, ...$args) method using match() to construct the right one. Create one of each via the factory and echo their areas.

View solution

CHALLENGE 3

Create a SortStrategy interface with a sort(array $items): array method, and two implementations: AscendingSort and DescendingSort. Write a class Report that accepts a SortStrategy via its constructor and a generate(array $items) method using it. Demonstrate swapping strategies on two different Report instances with the same data.

View solution

CHAPTER 2 QUICK REFERENCE

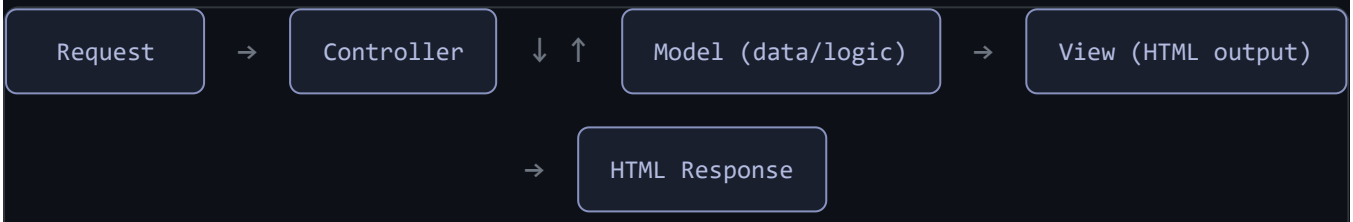
- **Singleton** — private constructor + static getInstance(); exactly one shared instance; use sparingly, prefer dependency injection generally
- **Factory** — a dedicated method/class for creating objects, decoupling calling code from specific classes
- **Strategy** — swappable behaviour behind a common interface, injected via the constructor
- **Repository** — hides data-storage details behind a simple, interface-based API; enables swapping real DB for a fake in tests
- **match expression** — strict comparison, no break needed, returns a value directly
- **Next chapter:** PHP and MVC architecture — building a simple framework from scratch

COURSE 3 · CH 3

PHP and MVC Architecture: Building a Simple Framework From Scratch

The pattern behind almost every modern PHP framework, built up piece by piece to see exactly what it's solving

The Intermediate capstone's blog put a fair amount of logic directly inside files like `index.php` and `new_post.php`. **MVC** (Model-View-Controller) separates an application into three distinct responsibilities, routed through a single entry point — the architecture underneath Laravel, Symfony, and most other PHP frameworks (previewed properly in Chapter 6).



Controller receives the request, asks the Model for data, hands it to a View to render — never the other way around

The Three Roles

Model

Data and business logic; in this course, the Post class from the Intermediate capstone is already a Model

View

Purely the HTML/output template; no business logic, no database calls

Controller

Receives the request, talks to the Model, picks a View, passes data to it; deliberately "thin"

The Front Controller — One Entry Point for Everything

Instead of one PHP file per page (as the Intermediate blog used), every request is routed through a single file, which decides what to do based on the URL.

```
mini-framework/
├── public/
│   └── index.php          ← the ONE entry point, the front controller
└── src/
    ├── Router.php
    ├── Controllers/
    │   └── PostController.php
    ├── Models/
    │   └── Post.php
    └── views/
        └── post_list.php
```

A Minimal Router

src/Router.php

```php
<?php
  namespace App;

  class Router {
      private array $routes = [];

      public function get(string $path, array $handler) {
          $this->routes[$path] = $handler;   // [ControllerClass, 'methodName']
      }

      public function dispatch(string $uri) {
          if (!isset($this->routes[$uri])) {
              http_response_code(404);
              echo "404 Not Found";
              return;
          }
          [$controllerClass, $method] = $this->routes[$uri];
          $controller = new $controllerClass();
          $controller->$method();
      }
  }
?>
```

`new $controllerClass()` demonstrates that a class name doesn't have to be written literally — it can be stored in a variable and instantiated dynamically, which is exactly what a router needs to do without an enormous hardcoded if/elseif chain for every possible URL.

The Front Controller Itself

`public/index.php`

```php
<?php
  require __DIR__ . '/../vendor/autoload.php';

  use App\Router;
  use App\Controllers\PostController;

  $router = new Router();
  $router->get('/posts', [PostController::class, 'index']);

  $uri = strtok($_SERVER['REQUEST_URI'], '?');   // strip any ?query=string before matchi
  $router->dispatch($uri);
?>
```

::CLASS — GETTING A CLASS'S FULLY-QUALIFIED NAME AS A STRING, SAFELY

`PostController::class` resolves to the string `"App\Controllers\PostController"` at compile time — safer and more refactor-friendly than writing that string by hand, since renaming the class or its namespace would automatically update everywhere `::class` is used too.

A Thin Controller

`src/Controllers/PostController.php`

```php
<?php
  namespace App\Controllers;

  use App\Models\Post;

  class PostController {
      public function index() {
          $postModel = new Post(getDbConnection());
```

```
            $posts = $postModel->all();
            require __DIR__ . '/../../views/post_list.php';   // $posts is now available ins
        }
    }
?>
```

The controller's job is deliberately small: get the data, hand it to a view. No SQL, no HTML — both belong elsewhere. This is exactly the "thin controller, fat model" guidance frameworks generally encourage.

A Pure View — No Logic, Just Output

views/post_list.php

```
<?php foreach ($posts as $post): ?>
  <h2><?= htmlspecialchars($post['title']) ?></h2>
<?php endforeach; ?>
```

IF A VIEW STARTS CONTAINING DATABASE QUERIES OR BUSINESS LOGIC, THE SEPARATION HAS BROKEN DOWN

The whole benefit of MVC depends on each piece staying within its role — a view that queries the database directly, or a controller that builds raw SQL, has quietly become a single tangled mess again, just split across more files than before.

Coding Challenges

CHALLENGE 1

Add a second route to the Router example: GET /about, mapped to a new AboutController with a show() method that requires a simple views/about.php containing static "About us" text (no model needed). Trace through, in writing, exactly what happens from the request hitting index.php to the final HTML being produced.

View solution

CHALLENGE 2

Identify the MVC violation in this controller method, and rewrite it correctly: public function show() { $stmt = $pdo->query("SELECT * FROM posts"); echo "
"; foreach ($stmt as $row) { echo "
{$row['title']}
"; } echo "
"; } — explain in a comment what's wrong and how your rewrite fixes it.

📄 View solution

CHALLENGE 3

Extend the Router class to support route parameters, e.g. registering "/posts/{id}" and matching a real URL like "/posts/42" by extracting 42 and passing it as an argument to the controller method. Use a regular expression (Intermediate Chapter 9) to implement the matching.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **Model** — data and business logic; **View** — pure output template; **Controller** — thin, coordinates the two
- **Front controller** — one entry point (public/index.php) routes every request
- **Router** — maps a URL to a [ControllerClass, 'method'] pair, dispatches dynamically
- **new $className()** — instantiating a class from a variable, essential for dynamic dispatch
- **ClassName::class** — gets the fully-qualified class name as a refactor-safe string
- "Thin controller, fat model" — keep business logic and SQL out of controllers and views entirely
- **Next chapter:** testing with PHPUnit — unit tests, mocking

COURSE 3 · CH 4

Testing With PHPUnit: Unit Tests, Mocking

Automatically verifying code behaves correctly, and isolating it from real dependencies like a database while doing so

Every challenge across this entire series has been verified by reading expected output and reasoning about the code. PHPUnit automates that verification — writing assertions once, then re-running them in seconds, every time the code changes, to catch regressions immediately rather than discovering them later.

Installing PHPUnit

```
$ composer require --dev phpunit/phpunit
# installed under "require-dev" (Chapter 1) — a testing tool, not a production dependency
```

A First Unit Test

src/Calculator.php

```php
<?php
  class Calculator {
      public function add(float $a, float $b): float {
          return $a + $b;
      }
  }
?>
```

tests/CalculatorTest.php

```php
<?php
  use PHPUnit\Framework\TestCase;

  class CalculatorTest extends TestCase {
```

```
        public function testAddReturnsCorrectSum() {
            $calculator = new Calculator();
            $result = $calculator->add(2, 3);
            $this->assertEquals(5, $result);
        }
    }
?>
```

```
$ ./vendor/bin/phpunit tests/CalculatorTest.php
PHPUnit 10.x

.                                                                   1 / 1 (100%)

OK (1 test, 1 assertion)
```

A test class `extends TestCase` (Intermediate Chapter 2's inheritance, applied here). Each method starting with `test` is run automatically, and `$this->assertEquals(expected, actual)` checks the result matches — exactly the same exercise as every challenge's "expected output" comment throughout this course, just executed by PHPUnit instead of read by eye.

Common Assertions

ASSERTION	CHECKS
`assertEquals($expected, $actual)`	Values are equal (loose comparison)
`assertSame($expected, $actual)`	Values AND types match (strict, like ===)
`assertTrue() / assertFalse()`	A value is exactly true / false
`assertCount($count, $array)`	An array has exactly $count elements
`assertInstanceOf($class, $obj)`	An object is an instance of a given class
`expectException($class)`	The tested code throws a specific exception

Testing That an Exception Is Thrown

```php
<?php
  public function testDivideByZeroThrowsException() {
      $calculator = new Calculator();
      $this->expectException(DivisionByZeroError::class);
      $calculator->divide(10, 0);   // the test PASSES if this line throws; FAILS if it do
  }
?>
```

This directly tests the kind of behaviour built in Intermediate Chapter 3 — confirming the right exception type is genuinely thrown under the right conditions, automatically, rather than manually re-checking by eye every time the code changes.

The Problem Mocking Solves

The Intermediate capstone's `Post` class depends on a real `PDO` connection. Testing it directly would mean hitting an actual database — slow, and dependent on test data that could change. A **mock** is a fake stand-in object that behaves exactly as instructed, without needing the real dependency at all.

Mocking a Dependency

```php
<?php
  use PHPUnit\Framework\TestCase;

  class PostControllerTest extends TestCase {
      public function testGreetUserUsesRepositoryCorrectly() {
          // Create a fake UserRepository (Chapter 2's interface), no real DB involved at
          $mockRepo = $this->createMock(UserRepository::class);
          $mockRepo->method('find')
              ->willReturn(['name' => 'Philip']);   // tell the mock exactly what to retur

          ob_start();
          greetUser($mockRepo, 1);
          $output = ob_get_clean();

          $this->assertEquals("Hello, Philip", $output);
      }
```

```
    }
?>
```

`createMock(UserRepository::class)` generates a fake object implementing that interface. `->method('find')->willReturn([...])` programs exactly what calling `find()` on it should produce — no real database, no real network call, completely predictable and fast. `greetUser()` from Chapter 2 never needed to be written knowing it would be tested this way — it already depended only on the `UserRepository` interface, which is precisely what makes mocking it possible at all.

THIS IS EXACTLY WHY CHAPTER 2'S REPOSITORY PATTERN MATTERED

Code written against a concrete `PDO` object directly is much harder to mock cleanly. Code written against an interface (`UserRepository`) can have any implementation substituted in — a real one in production, a fake/mock one in tests — without the calling code needing to change or even know the difference.

Running the Whole Test Suite

```
$ ./vendor/bin/phpunit tests/
# or, using a configured composer script (Chapter 1):
$ composer test
```

Coding Challenges

CHALLENGE 1

Write a class StringHelper with a method reverse(string $s): string (using strrev()) and a method isPalindrome(string $s): bool. Write a PHPUnit test class with at least three test methods covering: a normal reverse, a true palindrome case, and a false (non-palindrome) case.

View solution

CHALLENGE 2

Using the Chapter 3 InsufficientFundsException-style BankAccount class from Intermediate Chapter 3, write a PHPUnit test that uses expectException() to confirm withdrawing more than the balance throws InsufficientFundsException, and a second test confirming a valid withdrawal correctly reduces the balance.

View solution

CHALLENGE 3

Using the Strategy pattern's ShoppingCart from Chapter 2, write a test that uses createMock() to create a fake DiscountStrategy whose apply() method is programmed (via willReturn) to always return 50, then asserts that ShoppingCart's checkout() correctly returns 50 regardless of the price passed in. Explain in a comment why this test never touches PercentageDiscount or FixedAmountDiscount at all.

View solution

CHAPTER 4 QUICK REFERENCE

- **composer require --dev phpunit/phpunit** — install as a dev dependency
- **extends TestCase**, methods starting with "test" are run automatically
- **assertEquals / assertSame / assertTrue / assertCount / assertInstanceOf** — common assertions
- **expectException(ClassName::class)** — confirms specific code throws the expected exception
- **createMock(InterfaceName::class)** — a fake stand-in object, no real dependency needed
- **->method('name')->willReturn(value)** — programs a mock's behaviour
- Code written against interfaces is far easier to mock than code tied directly to a concrete class
- **Next chapter:** security in depth — CSRF, XSS prevention, password hashing

CHAPTER 5: SECURITY IN DEPTH - CSRF, XSS, PASSWORD HASHING

COURSE 3 · CH 5

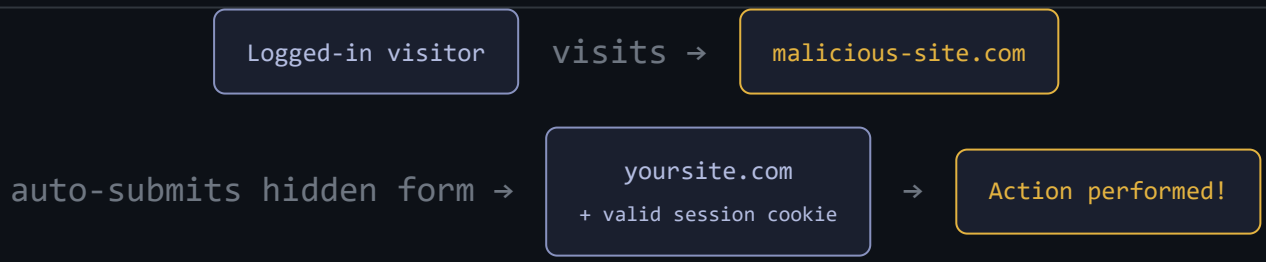
Security in Depth: CSRF, XSS Prevention, Password Hashing

Closing the last major gap left open since Intermediate — forged requests from a different site entirely

XSS (Intermediate Chapter 8) and password hashing (Intermediate Chapter 5) have already been covered properly. This chapter revisits both briefly as a security-focused recap, then introduces **CSRF** — the one major web vulnerability category not yet addressed anywhere in this series.

CSRF — Cross-Site Request Forgery

Imagine a logged-in visitor (with a valid session, Intermediate Chapter 5) visits a malicious page that auto-submits a hidden form to `yoursite.com/transfer-money`. Because the visitor's browser automatically attaches their session cookie to any request to that domain, the malicious request looks completely legitimate to the server — even though the visitor never intended to submit it.



The session cookie alone isn't enough proof the visitor actually intended this specific request

The Fix — CSRF Tokens

```php
<?php
  session_start();
  if (empty($_SESSION['csrf_token'])) {
      $_SESSION['csrf_token'] = bin2hex(random_bytes(32));   // a long, unguessable rando
```

```
  }
?>
<form method="post" action="transfer.php">
  <input type="hidden" name="csrf_token" value="<?= $_SESSION['csrf_token'] ?>">
  <!-- real form fields here -->
  <input type="submit">
</form>
```

transfer.php

```
<?php
  session_start();
  $submittedToken = $_POST['csrf_token'] ?? '';

  if (!hash_equals($_SESSION['csrf_token'] ?? '', $submittedToken)) {
      http_response_code(403);
      die("Invalid request.");
  }
  // token is valid — proceed with the actual transfer logic
?>
```

The token is embedded in the legitimate form, and re-checked on submission. The malicious page from the diagram above has no way to know or guess this token — it isn't part of the session cookie the browser sends automatically, so the forged request fails the check.

HASH_EQUALS() — COMPARING TOKENS, NOT JUST ==

`hash_equals()` performs a "timing-safe" comparison — a regular `===` string comparison can, in theory, take very slightly different amounts of time depending on how many leading characters match, which an attacker could measure over many attempts to slowly guess a secret value. `hash_equals()` is specifically designed to take the same amount of time regardless, which matters for comparing security tokens (though, with a 32-byte random token, brute-forcing is already practically infeasible regardless).

XSS Prevention — A Quick Recap

```
<?php
  echo htmlspecialchars($userInput);   // escape on EVERY output, every time (Intermediate
```

```
?>
```

Nothing new here beyond Intermediate Chapter 8's rule — included as a reminder that CSRF and XSS are entirely separate vulnerabilities requiring entirely separate fixes; defending against one does nothing for the other.

Password Hashing — A Quick Recap, Plus One Addition

```php
<?php
  $hash = password_hash($password, PASSWORD_DEFAULT);   // Intermediate Chapter 5

  // NEW: checking if a hash needs rehashing (e.g. PHP's default algorithm/cost improved)
  if (password_verify($password, $storedHash) && password_needs_rehash($storedHash, PASSW
      $newHash = password_hash($password, PASSWORD_DEFAULT);
      // save $newHash to the database, replacing $storedHash
  }
?>
```

`password_needs_rehash()` detects when a stored hash was created with weaker settings than PHP's current default — checked at login time (when the plain password is briefly available anyway), letting old hashes be quietly upgraded over time as users log in, without forcing a mass password reset.

A Combined Security Checklist

- ✓ Escape every piece of output with htmlspecialchars() — prevents XSS
- ✓ Use prepared statements for every database query — prevents SQL injection
- ✓ Include and verify a CSRF token on every state-changing form — prevents CSRF
- ✓ Hash passwords with password_hash(), verify with password_verify() — never store plain text
- ✓ Validate and sanitise file uploads, never trust the claimed MIME type — prevents malicious uploads
- ✓ Always call exit immediately after a header() redirect — prevents protected content leaking

Coding Challenges

CHALLENGE 1

Write a complete CSRF-protected form flow: a page generating and embedding a token, and a processing script that rejects the request with a 403 status if the token is missing or doesn't match, using hash_equals(). Test it conceptually by describing what would happen if an attacker's page tried to submit the same form action directly, without the token.

View solution

CHALLENGE 2

Write a function verifyAndUpgradePassword(string $password, string &$storedHash): bool that returns false immediately if the password doesn't verify, otherwise checks password_needs_rehash() and updates $storedHash (passed by reference) with a freshly generated hash if needed, then returns true. Explain in a comment why $storedHash needs to be passed by reference here.

View solution

CHALLENGE 3

Review this code and list every security issue you can find, then write a corrected version: session_start(); $name = $_POST['name']; echo "<form method='post'><input name='name' value='$name'><input type='submit'></form>"; $stmt = $pdo->query("SELECT * FROM users WHERE name = '$name'");

View solution

CHAPTER 5 QUICK REFERENCE

- **CSRF** — a forged request riding on a visitor's valid session, from a different malicious site
- **CSRF token** — a random, unguessable value embedded in forms and re-checked on submission
- **hash_equals()** — timing-safe comparison, used for comparing security tokens
- **htmlspecialchars()** — XSS prevention, on every output (Intermediate Chapter 8 recap)

- **password_hash() / password_verify()** — never store plain-text passwords (Intermediate Chapter 5 recap)
- **password_needs_rehash()** — detects outdated hash settings, allows quiet upgrades at login time
- CSRF, XSS, and SQL injection are three SEPARATE vulnerabilities — each needs its own specific defence
- **Next chapter:** working with an existing framework — Laravel/Symfony basics

COURSE 3 · CH 6

Working With an Existing Framework: Laravel/Symfony Basics

Recognising everything Chapters 1-5 built, now in the shape of a real, production-grade framework

Chapter 3's mini-framework deliberately built a Router, Controllers, Models, and Views from scratch to see exactly what each piece does. Laravel and Symfony are full, mature implementations of the same ideas, plus a great deal more. This chapter maps what's already been learned onto Laravel specifically (the more common starting point for new PHP developers), with Symfony noted as the main alternative.

Laravel vs Symfony — A Brief Comparison

Laravel

- Opinionated — one "right way" to do most things
- Extensive built-in tooling (auth, queues, mail)
- Generally faster to start a new project with
- Eloquent ORM by default

Symfony

- More modular — pick only the components needed
- Often used as the foundation OTHER frameworks (including parts of Laravel) build on
- Favoured for large, long-lived enterprise applications
- Doctrine ORM by default

Mapping This Course's Concepts Onto Laravel

THIS COURSE	LARAVEL'S EQUIVALENT
`Chapter 3's Router`	routes/web.php, routes/api.php
`Chapter 3's Controller`	app/Http/Controllers/

THIS COURSE	LARAVEL'S EQUIVALENT
`Intermediate's PDO + Post class`	Eloquent ORM models (app/Models/)
`Plain PHP view files`	Blade templates (resources/views/*.blade.php)
`Composer's vendor/autoload.php`	Same — Laravel IS a Composer package itself
`Chapter 2's Singleton/Factory patterns`	Laravel's "Service Container" — built-in dependency injection
`Chapter 5's CSRF tokens`	Built-in automatically via @csrf in Blade + VerifyCsrfToken middleware
`Chapter 4's PHPUnit tests`	Same PHPUnit, with extra Laravel-specific test helpers

This mapping is the entire point of this chapter: nothing here is conceptually new — Laravel mostly automates and polishes work this course has already done by hand.

Routes

`routes/web.php`

```php
<?php
  use App\Http\Controllers\PostController;
  use Illuminate\Support\Facades\Route;

  Route::get('/posts', [PostController::class, 'index']);
  Route::get('/posts/{id}', [PostController::class, 'show']);
?>
```

Recognisably similar to Chapter 3's `$router->get('/posts', [...])` — Laravel's `Route::get()` does the same job, with the `{id}` placeholder syntax matching Chapter 3's final challenge exactly.

Controllers

`app/Http/Controllers/PostController.php`

```php
<?php
  namespace App\Http\Controllers;
```

```
    use App\Models\Post;

    class PostController extends Controller {
        public function index() {
            $posts = Post::all();   // Eloquent — no SQL, no PDO calls written at all
            return view('posts.index', ['posts' => $posts]);
        }
    }
?>
```

`Post::all()` replaces the hand-written `SELECT * FROM posts` from Intermediate's capstone entirely — Eloquent generates the SQL automatically based on the model's name and a small amount of convention.

Eloquent — An ORM, Compared to Plain PDO

```
<?php
    namespace App\Models;
    use Illuminate\Database\Eloquent\Model;

    class Post extends Model {
        // that's it — no constructor, no SQL written anywhere
    }

    // Equivalent operations, Eloquent vs Intermediate's hand-written Post class:
    Post::find(1);                                  // vs $post->find(1) with manual PDO
    Post::where('title', 'LIKE', '%php%')->get();  // vs hand-writing a prepared statement
    Post::create(['title' => 'New', 'body' => '...']);  // vs $post->create(...)
?>
```

An **ORM** (Object-Relational Mapper) automates the pattern from Intermediate Chapter 4 and this course's Repository pattern — mapping database rows to objects and back, without hand-writing SQL for routine operations. Eloquent still uses prepared statements internally; the SQL-injection protections from Intermediate Chapter 4 are never bypassed, just generated automatically instead.

Blade — Laravel's Templating Engine

resources/views/posts/index.blade.php

```
<ul>
@foreach ($posts as $post)
    <li>{{ $post->title }}</li>
@endforeach
</ul>
```

{{ $VARIABLE }} AUTOMATICALLY ESCAPES OUTPUT

Blade's `{{ }}` syntax calls `htmlspecialchars()` automatically behind the scenes — the XSS-prevention discipline from Intermediate Chapter 8 is baked into the template syntax itself, rather than needing to be remembered and applied manually on every single echoed value.

Why Frameworks Still Matter, Having Built One From Scratch

- → Security fixes (CSRF, session handling) are maintained by a large community, not one developer
- → Conventions mean another Laravel developer can understand a new project quickly
- → A vast ecosystem of pre-built packages exists for almost anything (payments, search, file storage)
- → Time is spent on the actual application's logic, not re-solving routing/ORM/templating from scratch every time

CHAPTER 3'S MINI-FRAMEWORK WASN'T WASTED EFFORT

Understanding exactly what a Router, Controller, Model, and View each do — built by hand — is precisely what makes a real framework's structure and conventions make sense immediately, rather than feeling like arbitrary magic to memorise.

Coding Challenges

CHALLENGE 1

For each of these Chapter 3 mini-framework pieces, write out the corresponding Laravel equivalent and a one-sentence explanation: (a) $router->get('/about', [...]), (b) require

__DIR__.'/../views/post_list.php', (c) the Post class's hand-written find() method using PDO prepared statements.

View solution

CHALLENGE 2

Write a Blade template fragment that loops over an array of $products (each with 'name' and 'price' keys) and displays them in a list, using Blade's @foreach/@endforeach and {{ }} syntax. Explain in a comment exactly what security protection {{ }} provides automatically that this course's plain-PHP views always had to apply manually.

View solution

CHALLENGE 3

Write a short comparison (4-6 sentences) of building the Intermediate capstone blog by hand versus building the same blog in Laravel using Eloquent and Blade — covering what would be faster, what would be more educational, and one specific risk of relying entirely on a framework without understanding what it's doing underneath.

View solution

CHAPTER 6 QUICK REFERENCE

- **Laravel** — opinionated, batteries-included, Eloquent ORM, Blade templates
- **Symfony** — modular, component-based, often underlies other frameworks, Doctrine ORM
- **Route::get('/path', [Controller::class, 'method'])** — maps directly onto Chapter 3's Router
- **Eloquent** (Model::all(), ::find(), ::create()) — an ORM automating Intermediate Chapter 4's PDO patterns
- **Blade's {{ $var }}** — auto-escapes output, baking in Intermediate Chapter 8's XSS discipline
- **@csrf in Blade** — automates Chapter 5's CSRF token pattern
- Frameworks automate, not replace, the concepts already learned — that's why building one from scratch first matters

- **Next chapter:** building a REST API in PHP — routing, JSON responses, status codes

COURSE 3 · CH 7

Building a REST API in PHP: Routing, JSON Responses, Status Codes

Adapting Chapter 3's mini-framework to serve data instead of HTML pages

A REST API serves the same kind of application logic as a normal web page — just returning JSON (Intermediate Chapter 7) instead of HTML, with HTTP methods and status codes carrying meaning that a typical web form-based app barely uses. This chapter adapts Chapter 3's Router/Controller structure directly into an API.

REST's Core Idea — HTTP Methods Map to CRUD

GET /posts	**GET /posts/{id}**	**POST /posts**	**PUT /posts/{id}**	**DELETE /posts/{id}**
Read — list all posts	Read — one specific post	Create — a new post	Update — replace a post	Delete — remove a post

This is exactly Intermediate Chapter 4's CRUD, expressed through HTTP methods rather than separate page names like `new_post.php` or `delete_post.php`.

Returning JSON Instead of HTML

```php
<?php
  function jsonResponse($data, int $statusCode = 200) {
      http_response_code($statusCode);
      header('Content-Type: application/json');
      echo json_encode($data);
      exit;
  }

  jsonResponse(['id' => 1, 'title' => 'Hello']);
?>
```

A REST API needs the `Content-Type: application/json` header (matching Intermediate Chapter 7's requirement when sending JSON via cURL) so client applications know to parse the response as JSON, not HTML. A small helper like this avoids repeating those three lines in every controller method.

HTTP Status Codes — Communicating What Happened

CODE	MEANING
`200 OK`	Success — GET, PUT succeeded
`201 Created`	Success — POST created a new resource
`204 No Content`	Success — DELETE succeeded, nothing to return
`400 Bad Request`	The request itself is malformed/invalid
`401 Unauthorized`	Authentication required, and missing/invalid
`403 Forbidden`	Authenticated, but not allowed to do this
`404 Not Found`	The requested resource doesn't exist
`422 Unprocessable Entity`	Valid request format, but failed validation
`500 Internal Server Error`	Something broke on the server's end

> **RETURNING 200 FOR EVERY RESPONSE, EVEN ERRORS, IS A COMMON AND GENUINELY UNHELPFUL API MISTAKE**
>
> A client application relies on status codes to decide how to react — retry, show a specific error, redirect to login. An API that always returns 200 regardless of what actually happened forces every client to additionally inspect the response body just to find out if something went wrong, defeating much of the purpose of having status codes at all.

An API Controller, Built on Chapter 3's Pattern

```
<?php
  namespace App\Controllers;
  use App\Models\Post;
```

```
use App\Models\PostNotFoundException;

class PostApiController {
    public function index() {
        $post = new Post(getDbConnection());
        jsonResponse($post->all());
    }

    public function show(string $id) {
        $post = new Post(getDbConnection());
        try {
            jsonResponse($post->find((int)$id));
        } catch (PostNotFoundException $e) {
            jsonResponse(['error' => 'Post not found'], 404);
        }
    }

    public function store() {
        $input = json_decode(file_get_contents('php://input'), true);
        if (empty($input['title']) || empty($input['body'])) {
            jsonResponse(['error' => 'title and body are required'], 422);
        }
        $post = new Post(getDbConnection());
        $newId = $post->create($input['title'], $input['body']);
        jsonResponse(['id' => $newId], 201);
    }
}
?>
```

`file_get_contents('php://input')` reads the raw request body — a JSON API client sends a JSON-encoded body rather than a normal HTML form, so `$_POST` (used throughout this entire course up to now) doesn't get populated; the body has to be read and decoded manually instead.

> **POSTNOTFOUNDEXCEPTION, WRITTEN BACK IN INTERMEDIATE CHAPTER 3, SLOTS STRAIGHT INTO THIS API CONTROLLER**
>
> The exception itself never needed to be written knowing it would eventually power an API response — catching it here and converting it into a 404 JSON response is a perfect illustration of well-designed exceptions being reusable across very different contexts (a web page showing "not found" text, vs. an API returning a 404 status).

Routing API Requests by Method, Not Just Path

```php
<?php
  $method = $_SERVER['REQUEST_METHOD'];
  $uri = strtok($_SERVER['REQUEST_URI'], '?');
  $controller = new PostApiController();

  match (true) {
      $method === 'GET' && $uri === '/posts'  => $controller->index(),
      $method === 'POST' && $uri === '/posts' => $controller->store(),
      default => jsonResponse(['error' => 'Not found'], 404),
  };
?>
```

Unlike Chapter 3's Router (which only mapped by path), an API router must check the HTTP method too — `GET /posts` and `POST /posts` share the same path but mean entirely different operations.

Coding Challenges

CHALLENGE 1

Write a jsonResponse() helper function (as shown in the chapter), then write a small script simulating a "user not found" scenario: it should call jsonResponse(['error' => 'User not found'], 404). Explain in a comment why setting the status code AND the Content-Type header both matter, even though the JSON body itself is valid either way.

View solution

CHALLENGE 2

Write a destroy(string $id) method for PostApiController that deletes a post by ID and returns a 204 status with an empty body on success, or a 404 JSON error if the post doesn't exist (using the Post class's find() method first to check, catching PostNotFoundException).

View solution

CHALLENGE 3

Write a store() method (like the chapter's example) that additionally validates the title is no longer than 200 characters, returning 422 with a specific error message if it is. Then write the matching cURL-based test code (Intermediate Chapter 7) that POSTs a JSON body to this endpoint and checks the response status code and decoded body.

View solution

CHAPTER 7 QUICK REFERENCE

- **HTTP methods map to CRUD:** GET (read), POST (create), PUT (update), DELETE (delete)
- **jsonResponse()** — sets status code + Content-Type, json_encode()'s the body, exits
- **200/201/204** — success variants; **400/401/403/404/422** — client error variants; **500** — server error
- **file_get_contents('php://input')** — reads the raw JSON request body; $_POST is NOT populated for JSON requests
- Route by method AND path — GET /posts and POST /posts are different operations on the same path
- Existing exceptions/models from earlier chapters slot directly into an API — no rewriting needed, just a different response format
- **Next chapter:** asynchronous/concurrent patterns in PHP — queues, background jobs

COURSE 3 · CH 8

Asynchronous/Concurrent Patterns in PHP: Queues, Background Jobs

Handling slow work without making a visitor wait for it — and why PHP's request model makes this genuinely different from other languages

A typical PHP script runs from start to finish within a single request, then the process ends. Sending a welcome email, generating a large PDF report, or resizing an uploaded image can take seconds — far too slow to make a visitor wait for, and risky to do directly inside the request/response cycle at all.

PHP's Request Model — Why This Matters

Node.js / long-running processes

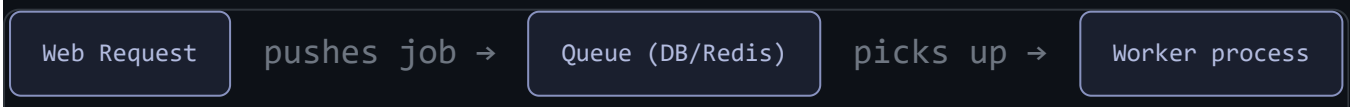
- One process stays alive, handling many requests over time
- Genuine in-process async/await is natural

Traditional PHP (with most web servers)

- Each request typically starts a fresh process/script execution
- No natural way to "keep working after the response is sent" reliably

This is exactly why PHP relies on a separate mechanism — a job queue, backed by a process that runs independently of any web request — rather than genuine in-request asynchronous code, for anything that needs to happen "in the background."

The Queue Pattern



The request returns immediately (response sent right away); a separate worker process handles the slow work whenever it gets to it

Instead of doing slow work directly, the request "pushes" a description of the job (what needs doing, with what data) onto a queue — typically a database table or Redis — and responds to

the visitor right away. A completely separate, long-running worker process continuously checks the queue and processes jobs as they appear.

A Minimal Database-Backed Queue

```php
<?php
  // Pushing a job — inside a normal web request, e.g. after user registration
  function queueEmail(PDO $pdo, string $to, string $subject) {
      $stmt = $pdo->prepare("INSERT INTO jobs (type, payload, status) VALUES ('send_email'
      $stmt->execute(['payload' => json_encode(['to' => $to, 'subject' => $subject])]);
  }

  // In the registration controller:
  queueEmail($pdo, $user['email'], 'Welcome!');
  jsonResponse(['message' => 'Registered successfully'], 201);   // responds immediately -
?>
```

The Worker — A Long-Running Separate Script

worker.php — run via CLI, kept running continuously (e.g. by supervisord/systemd)

```php
<?php
  while (true) {
      $stmt = $pdo->query("SELECT * FROM jobs WHERE status = 'pending' LIMIT 1");
      $job = $stmt->fetch();

      if ($job) {
          $payload = json_decode($job['payload'], true);
          try {
              sendEmail($payload['to'], $payload['subject']);   // the genuinely slow oper
              markJobComplete($pdo, $job['id']);
          } catch (Exception $e) {
              markJobFailed($pdo, $job['id'], $e->getMessage());   // Chapter 3-style exce
          }
      } else {
          sleep(2);   // nothing to do — wait briefly before checking again
      }
```

```
    }
?>
```

The worker runs entirely outside the web server — typically started once and kept alive indefinitely by a process supervisor — repeatedly polling for new jobs. This is a fundamentally different execution model from a normal web request, which is exactly why it needs to be designed and run separately.

> **A FAILED JOB MUST BE HANDLED DELIBERATELY, NOT SILENTLY DROPPED**
>
> Marking a job `'failed'` (rather than leaving it stuck as `'pending'` forever, or deleting it) preserves a record of what went wrong and allows a retry strategy to be built — exactly the kind of deliberate error handling from Intermediate Chapter 3, applied to a background process instead of a request.

Why Not Just Use a Separate Thread?

PHP doesn't have built-in multi-threading the way some other languages do (true threads sharing memory within one process). The queue pattern sidesteps this entirely — instead of threads sharing memory, completely separate PHP processes communicate only through the queue (a database row, or a message broker like Redis), which is simpler to reason about and naturally survives a worker crashing or restarting.

Real-World Tooling

This chapter's example is intentionally minimal to show the underlying mechanism clearly. Real projects typically use a dedicated queue library — Laravel's built-in Queue system (Chapter 6), or standalone packages — which add retry logic, delayed jobs, and proper worker process management on top of the same basic pattern demonstrated here.

Coding Challenges

CHALLENGE 1

Design (in writing) the SQL schema for a "jobs" table suitable for the queue pattern shown in this chapter — list each column, its type, and a one-sentence reason it's needed. Include at minimum: an id, a job type, a payload, a status, and a created timestamp.

View solution

CHALLENGE 2

Write a queueJob(PDO $pdo, string $type, array $payload) function that inserts a new pending job with a JSON-encoded payload, generalising the chapter's queueEmail() to handle any job type. Then show how it would be called for two different job types: 'send_email' and 'resize_image'.

View solution

CHALLENGE 3

Extend the worker loop from this chapter to dispatch to a different function depending on the job's "type" column (using match()), supporting both 'send_email' and 'resize_image' job types, each with its own try/catch and markJobComplete/markJobFailed calls. Explain in a comment what would happen if one job type's handler threw an uncaught exception, with no try/catch around the dispatch at all.

View solution

CHAPTER 8 QUICK REFERENCE

- **PHP's request model** — typically one script run per request, no natural in-process background work
- **Queue pattern** — push a job description, respond immediately, a separate worker processes it later
- **Jobs table / Redis** — common backing stores for a queue
- **Worker** — a long-running CLI script, kept alive by a process supervisor, polling for pending jobs
- **Failed jobs** — mark deliberately, don't silently drop; enables retry strategies
- No native threads — separate processes + a shared queue, rather than shared-memory threading
- Real projects use dedicated tooling (Laravel Queues, etc.) — this chapter shows the underlying mechanism

- **Next chapter:** performance — opcache, profiling, caching strategies

COURSE 3 · CH 9

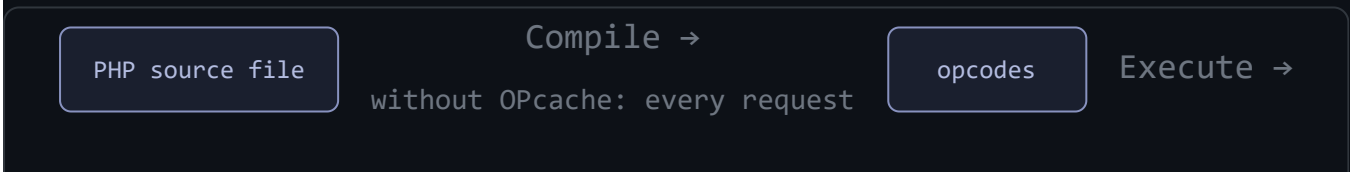
Performance: Opcache, Profiling, Caching Strategies

Making PHP fast in the ways that actually matter, and finding out where time is really being spent before optimising anything

Every chapter so far has focused on correctness, structure, and security. This chapter covers the other axis: making an application fast enough — and, more importantly, knowing exactly where it's slow before changing anything, rather than guessing.

OPcache — Caching Compiled PHP, Not Application Data

Every time a plain PHP script runs, PHP first compiles the source code into "opcodes" before executing them — a real cost, repeated on every single request, for files that haven't changed at all between requests.



OPcache stores the compiled opcodes in shared memory, skipping recompilation on every later request

```
# php.ini
opcache.enable=1
opcache.memory_consumption=128
opcache.max_accelerated_files=10000
opcache.revalidate_freq=2
```

OPCACHE CACHES COMPILED CODE, NEVER YOUR APPLICATION'S DATA

A genuinely common confusion: OPcache has nothing to do with caching database query results, API responses, or rendered HTML — it only avoids recompiling PHP source files that haven't changed. Application-level caching (covered below) is an entirely separate, additional concern.

Profiling — Finding Out Where Time Actually Goes

Guessing at performance problems is genuinely unreliable — the code that "feels slow" while reading it is frequently not where the real time is spent. A profiler measures actual execution time per function call, across a real run of the application.

```
# Xdebug's profiler (one common option), enabled via php.ini:
xdebug.mode=profile
xdebug.output_dir=/tmp/profiles
# produces a cachegrind file, viewable with a tool like KCachegrind/QCacheGrind,
# showing exactly which function calls consumed the most cumulative time
```

A SIMPLE, DEPENDENCY-FREE PROFILING TECHNIQUE: MANUAL TIMING

```php
<?php
  $start = microtime(true);
  $posts = $post->all();
  $elapsed = microtime(true) - $start;
  echo "Query took: " . round($elapsed * 1000, 2) . "ms";
?>
```

Wrapping a suspected slow section with `microtime(true)` before and after gives a quick, genuinely useful measurement without installing any tooling — a reasonable first step before reaching for a full profiler.

The N+1 Query Problem — A Classic Performance Bug

```php
<?php
  // BAD — one query to get posts, then ANOTHER query per post inside the loop
  $posts = $pdo->query("SELECT * FROM posts")->fetchAll();
  foreach ($posts as $post) {
      $stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
      $stmt->execute(['id' => $post['user_id']]);
      $author = $stmt->fetch();   // for 100 posts, this runs 100 EXTRA queries
  }

  // GOOD — one query total, using a JOIN
  $posts = $pdo->query("
      SELECT posts.*, users.name AS author_name
```

```
        FROM posts
        JOIN users ON users.id = posts.user_id
    ")->fetchAll();
?>
```

"N+1" refers to the pattern: 1 initial query, then N more queries (one per row) inside a loop — for 100 posts, that's 101 total database round-trips instead of 1. This is one of the single most common real-world PHP performance problems, and it's a structural fix (combining queries), not a "make PHP faster" fix.

Application-Level Caching

```
<?php
    function getPopularPosts(PDO $pdo): array {
        $cacheFile = '/tmp/popular_posts_cache.json';

        if (file_exists($cacheFile) && (time() - filemtime($cacheFile)) < 300) {
            return json_decode(file_get_contents($cacheFile), true);   // cache is fresh (ur
        }

        $posts = $pdo->query("SELECT * FROM posts ORDER BY views DESC LIMIT 10")->fetchAll()
        file_put_contents($cacheFile, json_encode($posts));
        return $posts;
    }
?>
```

A simple file-based cache here; real applications typically use Redis or Memcached for this purpose instead. The core idea is identical regardless of storage: run an expensive operation once, store the result, and serve the stored result to subsequent requests until it's considered stale.

LAYER	CACHES	EXAMPLE
OPcache	Compiled PHP opcodes	Automatic, set in php.ini
Application cache	Expensive computed results (queries, API responses)	Redis, Memcached, file-based
HTTP cache headers	Whole responses, at the browser/CDN level	Cache-Control, ETag headers

CACHING INTRODUCES A NEW PROBLEM: STALE DATA

A cache that's too "sticky" can show outdated information after the underlying data has genuinely changed — cache invalidation (clearing or updating a cache entry exactly when the relevant data changes) is often the harder half of any caching strategy, not the caching itself.

Coding Challenges

CHALLENGE 1

Explain in your own words the difference between what OPcache caches and what an application-level cache (like Redis) caches, using a concrete example of each kind of content being cached.

View solution

CHALLENGE 2

Identify the N+1 query problem in this code, and rewrite it using a single JOIN query instead: $comments = $pdo->query("SELECT * FROM comments")->fetchAll(); foreach ($comments as $c) { $stmt = $pdo->prepare("SELECT name FROM users WHERE id = :id"); $stmt->execute(['id' => $c['user_id']]); $author = $stmt->fetch(); }

View solution

CHALLENGE 3

Write a function getCachedOrCompute(string $cacheFile, int $ttlSeconds, callable $computeFn) that generalises the chapter's file-cache example — if a fresh cache file exists, return its decoded contents; otherwise call $computeFn(), cache its result, and return it. Show it being used to cache the result of an expensive function getExpensiveReport().

View solution

CHAPTER 9 QUICK REFERENCE

- **OPcache** — caches COMPILED PHP code (opcodes), not application data; configured in php.ini
- **Profiling** — measure before optimising; microtime(true) for quick checks, Xdebug for full profiles
- **N+1 query problem** — one query, then one MORE per row in a loop; fix with a JOIN
- **Application-level caching** — store an expensive result once, serve it until stale (Redis, Memcached, file-based)
- **Cache invalidation** — often the hardest part: deciding exactly when cached data becomes stale
- Three caching layers: OPcache (code), application cache (data), HTTP cache headers (whole responses)
- **Next chapter:** capstone — a small REST API with auth, database, and tests

CAPSTONE: A SMALL REST API WITH AUTH, DATABASE, TESTS

COURSE 3 · CAPSTONE

Capstone: A Small REST API With Auth, Database, and Tests

Every chapter of PHP Advanced, combined into one cohesive, testable API

This capstone brings the entire Advanced course together: a REST API (Chapter 7) backed by a Repository (Chapter 2), authenticated with tokens, tested with PHPUnit and mocking (Chapter 4), with CSRF/XSS/SQL-injection defences (Chapter 5) and basic dependency management (Chapter 1) — the shape of a genuinely real, small production API.

CH 1 composer.json + PSR-4	**CH 2** Repository pattern for tasks	**CH 3** Router + thin controllers
CH 4 PHPUnit + mocked repository	**CH 5** Token auth, hash_equals	**CH 7** JSON responses, status codes

What We're Building

A minimal "Task API" — authenticated CRUD over a list of tasks, each belonging to a user, with a token required for every request.

```
task-api/
├── composer.json          — PSR-4: "App\\" → "src/"
├── src/
│   ├── Router.php
│   ├── Controllers/
│   │   └── TaskController.php
│   ├── Models/
│   │   ├── TaskRepository.php          — interface (Ch 2)
│   │   ├── MySqlTaskRepository.php     — real implementation
│   │   └── TaskNotFoundException.php
│   └── Auth.php
├── tests/
```

```
│   └── TaskControllerTest.php          — uses a MOCKED repository
└── public/
    └── index.php
```

The Repository Interface and Implementation

src/Models/TaskRepository.php

```php
<?php
  namespace App\Models;

  interface TaskRepository {
      public function all(): array;
      public function find(int $id): array;   // throws TaskNotFoundException
      public function create(string $title): int;
      public function delete(int $id): void;
  }
?>
```

src/Models/MySqlTaskRepository.php

```php
<?php
  namespace App\Models;

  class TaskNotFoundException extends \Exception {}

  class MySqlTaskRepository implements TaskRepository {
      public function __construct(private \PDO $pdo) {}

      public function all(): array {
          return $this->pdo->query("SELECT * FROM tasks")->fetchAll();
      }

      public function find(int $id): array {
          $stmt = $this->pdo->prepare("SELECT * FROM tasks WHERE id = :id");
          $stmt->execute(['id' => $id]);
          $task = $stmt->fetch();
          if (!$task) {
              throw new TaskNotFoundException("No task with ID $id");
          }
          return $task;
      }
```

```
        public function create(string $title): int {
            $stmt = $this->pdo->prepare("INSERT INTO tasks (title) VALUES (:title)");
            $stmt->execute(['title' => $title]);
            return (int)$this->pdo->lastInsertId();
        }

        public function delete(int $id): void {
            $stmt = $this->pdo->prepare("DELETE FROM tasks WHERE id = :id");
            $stmt->execute(['id' => $id]);
        }
    }
?>
```

Simple Token Authentication

src/Auth.php

```
<?php
    namespace App;

    class Auth {
        private const VALID_TOKEN = 'a1b2c3d4e5f6';   // in real use: per-user tokens, store

        public static function check(): bool {
            $header = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
            $token = \str_replace('Bearer ', '', $header);
            return \hash_equals(self::VALID_TOKEN, $token);   // Chapter 5's timing-safe com
        }
    }
?>
```

A request is expected to include `Authorization: Bearer a1b2c3d4e5f6`. This is deliberately simplified for the capstone's scope — real APIs typically use per-user tokens (often JWTs) issued at login and stored hashed, never a single hardcoded shared secret.

The Controller — Tying Auth, Repository, and JSON Together

src/Controllers/TaskController.php

```php
<?php
  namespace App\Controllers;
  use App\Auth;
  use App\Models\TaskRepository;
  use App\Models\TaskNotFoundException;

  class TaskController {
      public function __construct(private TaskRepository $tasks) {}   // depends on the IN

      private function requireAuth() {
          if (!Auth::check()) {
              jsonResponse(['error' => 'Unauthorized'], 401);
          }
      }

      public function index() {
          $this->requireAuth();
          jsonResponse($this->tasks->all());
      }

      public function show(string $id) {
          $this->requireAuth();
          try {
              jsonResponse($this->tasks->find((int)$id));
          } catch (TaskNotFoundException $e) {
              jsonResponse(['error' => 'Task not found'], 404);
          }
      }

      public function store() {
          $this->requireAuth();
          $input = \json_decode(\file_get_contents('php://input'), true);
          if (empty($input['title'])) {
              jsonResponse(['error' => 'title is required'], 422);
          }
          $id = $this->tasks->create($input['title']);
          jsonResponse(['id' => $id], 201);
      }
  }
?>
```

Every endpoint calls `requireAuth()` first — Chapter 5's auth check, applied consistently. The constructor depends on the `TaskRepository` interface, not the concrete `MySqlTaskRepository`

— exactly what makes the test below possible without touching a real database at all.

Testing the Controller — With a Mocked Repository

tests/TaskControllerTest.php

```php
<?php
  use PHPUnit\Framework\TestCase;
  use App\Controllers\TaskController;
  use App\Models\TaskRepository;
  use App\Models\TaskNotFoundException;

  class TaskControllerTest extends TestCase {
      public function testShowReturns404ForMissingTask() {
          $mockRepo = $this->createMock(TaskRepository::class);
          $mockRepo->method('find')->willThrowException(new TaskNotFoundException());

          $controller = new TaskController($mockRepo);

          // (capturing jsonResponse's output/status code omitted for brevity —
          // see Chapter 4's ob_start()/ob_get_clean() pattern for the full technique)
          $this->expectOutputRegex('/Task not found/');
          $controller->show('999');
      }
  }
?>
```

No real database, no real authentication header — this test verifies `TaskController`'s own logic in complete isolation. This is the entire payoff of designing against an interface from the very start, demonstrated concretely rather than just described.

> **DELIBERATE SIMPLIFICATIONS FOR A CAPSTONE OF THIS SCOPE**
>
> The hardcoded single auth token, no rate limiting, and no pagination are all genuine gaps a production API would need to close — flagged here as known scope boundaries, not oversights, exactly matching the pattern from the Intermediate capstone.

Coding Challenge — Extend the Capstone

FINAL CHALLENGE

Add a destroy(string $id) method to TaskController (auth-checked, returns 204 on success or 404 via TaskNotFoundException), and write a corresponding PHPUnit test using a mocked TaskRepository whose delete() method is asserted to have been called exactly once with the correct ID, using $mockRepo->expects($this->once())->method('delete')->with(5);.

View solution

COURSE 3 COMPLETE — PHP ADVANCED

- Chapters 1-9 covered: Composer in depth, design patterns, MVC from scratch, PHPUnit/mocking, security in depth, performance/caching, framework overview, REST APIs, async/queues
- This capstone combined Repository, mocking, token auth, and REST conventions into one small, genuinely testable API
- Known, deliberate gaps: single hardcoded token rather than per-user auth, no rate limiting, no pagination
- **The PHP series (Fundamentals → Intermediate → Advanced) is now complete** — 29 chapters across 3 courses, each with coding challenges and worked solutions