PHP Programming Language: Definitive Guide from Hello World to Expert-Level Topics

PHP (Hypertext Preprocessor) is a widely-used, open-source, server-side scripting language designed for web development but also capable of general-purpose programming. Known for its simplicity, flexibility, and integration with HTML, PHP powers millions of websites, including WordPress, Laravel-based apps, and APIs. This README is the ultimate guide to PHP, covering every topic from beginner to expert level with exhaustive explanations, practical examples, edge cases, and cross-connections, mirroring the depth of a Rust README.

Table of Contents

Introduction to PHP

PHP, created by Rasmus Lerdorf in 1994, is a server-side language embedded in HTML, excelling in dynamic web content generation. Its key features include:

- **Server-Side Execution**: Runs on web servers (e.g., Apache, Nginx), generating HTML or JSON.
- **Dynamic Typing**: Flexible variable types without declarations.

- **Extensive Ecosystem**: Rich standard library and Composer for dependency management.
- **Web Focus**: Built-in support for HTTP, forms, sessions, and databases.
- **Multi-Paradigm**: Supports procedural, object-oriented, and functional programming.
- **Use Cases**: Web apps, APIs, content management systems (e.g., WordPress), e-commerce, and scripting.

PHP's popularity stems from its ease of use, integration with databases (MySQL, PostgreSQL), and frameworks like Laravel and Symfony.

Getting Started: Hello, World!

Install PHP via php.net or package managers (`apt`, `brew`). Use a web server (Apache/Nginx) or PHP's built-in server for development.

Basic "Hello, World!" script (`index.php`):

```
<?php
echo "Hello, World!";
?>
```

Run with PHP's built-in server:

```
php -S localhost:8000
```

Access at `http://localhost:8000`. Install Composer for dependency management:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

Create a project:

```
mkdir myproject
cd myproject
composer init
```

PHP Versions

PHP has evolved significantly:

- **PHP 5**: (2004–2018) Introduced OOP, PDO, and MySQLi.
- **PHP 7**: (2015–2022) Major performance improvements, scalar type hints, return types.
- **PHP 8**: (2020–present) JIT, attributes, union types, match expressions, and more.

Key features by version:

- **7.0**: Scalar type declarations, return types, null coalescing operator (`??`).
- **7.1**: Nullable types, `void` return, iterable pseudo-type.
- **7.2**: Parameter type widening, sodium cryptography.
- **7.3**: Flexible heredoc/nowdoc, trailing commas in function calls.
- **7.4**: Arrow functions, typed properties, FFI.

- **8.0**: JIT, attributes, union types, named arguments, match expression.
- **8.1**: Enums, fibers, readonly properties, first-class callables.
- **8.2**: Readonly classes, standalone types (`null`, `true`, `false`).
- **8.3**: Typed class constants, dynamic class constant fetch.
- **8.4**: (October 2024) Asymmetric visibility, new array functions, improved FFI.

Check version:

```
php -v
```

Use `phpbrew` or Docker for version management:

```
phpbrew install 8.4
```

Basic Concepts

Variables and Data Types

PHP uses dynamic typing with `$` for variable names. Types:

- **Scalar**: `int`, `float`, `string`, `bool`.
- **Compound**: `array`, `object`, `callable`, `iterable`.
- **Special**: `null`, `resource`, `mixed` (8.0+).

```
<?php
$int = 42;
$float = 3.14;
$string = "Hello";
$bool = true;
$null = null;
$array = [1, 2, 3];
$object = new stdClass();

var_dump($int); // int(42)
echo gettype($string); // string

// Type juggling
$sum = "5" + 3; // 8 (string cast to int)
$concat = "5" . 3; // "53" (int cast to string)
```

Edge Cases:

- **Type Juggling**: `"1e3" == 1000` due to scientific notation.
- **Loose Comparison**: `0 == "0"`, but `0 !== "0"`; use `===`.
- **Float Precision**: `0.1 + 0.2 !== 0.3`; use `bcmath`.

Cross-Connections:

- **Arrays**: Primary data structure.

- **OOP**: Objects are instances of classes.
- **Functions**: Callables include closures.

Functions

Functions are defined with `function`, supporting default arguments and type hints.

```
<?php
function add(int $a, int $b = 0): int {
    return $a + $b;
}

function greet(string $name, string $greeting = "Hello"): string {
    return "$greeting, $name!";
}

// Variable arguments
function process(...$args): void {
    print_r($args);
}

// Arrow function (7.4+)
$double = fn($x) => $x * 2;

echo add(5, 3); // 8
echo greet("Alice"); // Hello, Alice!
process(1, 2, "foo"); // Array([0] => 1, [1] => 2, [2] => foo)
echo $double(5); // 10
```

Advanced Features:

- **Type Hints**: Scalar, union (8.0+), and nullable types.
- **Named Arguments**: (8.0+) `greet(name: "Alice")`.
- **First-Class Callables**: (8.1+) `$fn = strlen(...)`.

```
<?php
function compute(int|float $x, callable $fn): int|float {
    return $fn($x);
}

echo compute(5, fn($n): int => $n * 2); // 10
```

Edge Cases:

- **Type Coercion**: `add("5", 3)` coerces to `8` in non-strict mode.
- **Default Argument Scope**: Static defaults are evaluated once.
- **Callable Syntax**: Invalid callables throw `TypeError`.

Cross-Connections:

- **Functional Programming**: Closures and arrow functions.
- **OOP**: Methods are functions in classes.

- **Web Development**: Functions handle HTTP logic.

Control Flow

PHP supports `if`, `switch`, `for`, `foreach`, `while`, and `match` (8.0+).

```
<?php
$x = 7;
if ($x > 5) {
    echo "Greater";
} elseif ($x === 5) {
    echo "Equal";
} else {
    echo "Lesser";
}

// Loops
for ($i = 1; $i <= 3; $i++) {
    echo $i; // 1, 2, 3
}

$array = ["a" => 1, "b" => 2];
foreach ($array as $key => $value) {
    echo "$key: $value\n";
}

// Match (8.0+)
function describe($value): string {
    return match (true) {
        is_int($value) && $value > 0 => "Positive integer",
        is_string($value) => "String: $value",
        default => "Other",
    };
}

echo describe(42); // Positive integer
echo describe("hello"); // String: hello
```

Advanced Features:

- **Alternative Syntax**: `<?php if ($x): ?> ... <?php endif; ?>` for templates.
- **Match Expression**: Strict typing, exhaustiveness (8.0+).
- **Break/Continue Levels**: `break 2` exits nested loops.

Edge Cases:

- **Switch Fallthrough**: Cases without `break` continue.
- **Foreach by Reference**: Modifying arrays during iteration can cause bugs.
- **Match Type Safety**: Non-exhaustive `match` throws `UnhandledMatchError`.

Cross-Connections:

- **Arrays**: `foreach` iterates over arrays.

- **Error Handling**: Combine with `try`/`catch`.
- **Web Development**: Control flow in templates.

Data Structures

Arrays

Arrays are PHP's primary data structure, acting as lists, maps, or sets.

```
<?php
$list = [1, 2, 3];
$assoc = ["a" => 1, "b" => 2];
$list[] = 4; // Append
$assoc["c"] = 3;

print_r($list); // [1, 2, 3, 4]
print_r($assoc); // ["a" => 1, "b" => 2, "c" => 3]

// Array destructuring
[$x, $y] = [10, 20];
echo "$x, $y"; // 10, 20

// Array functions
$filtered = array_filter($list, fn($x) => $x % 2 === 0);
$mapped = array_map(fn($x) => $x * 2, $list);
echo implode(", ", $filtered); // 2, 4
echo implode(", ", $mapped); // 2, 4, 6, 8
```

Advanced Features:

- **Splat Operator**: `...$array` for unpacking.
- **ArrayAccess**: Classes implementing `ArrayAccess` act like arrays.
- **Short Array Syntax**: `[1, 2]` since 5.4.

Edge Cases:

- **Numeric Strings**: `$arr["1"]` and `$arr[1]` are equivalent.
- **Reference Issues**: `foreach ($arr as &$v)` retains references post-loop.
- **Memory Usage**: Large arrays consume significant memory; use generators for streaming.

Cross-Connections:

- **Functional Programming**: Array functions enable functional style.
- **OOP**: Arrays store object properties.
- **Web Development**: Arrays handle form data.

Strings

Strings are sequences of bytes (UTF-8 in modern PHP) with rich functions.

```php
<?php
$str = "Hello, World!";
echo strtolower($str); // hello, world!
echo explode(",", $str)[0]; // Hello

// Formatting
$name = "Alice";
$age = 30;
printf("%s is %d", $name, $age); // Alice is 30
echo "Name: $name, Age: $age"; // Name: Alice, Age: 30

// Heredoc/Nowdoc
$heredoc = <<<EOD
Hello, $name!
EOD;
echo $heredoc; // Hello, Alice!

// UTF-8
$smile = "😊";
echo mb_strlen($smile); // 1
echo strlen($smile); // 4 (bytes)
```

Advanced Features:

- **Multibyte Strings**: `mb_*` functions for UTF-8.
- **Regular Expressions**: `preg_match`, `preg_replace` for patterns.
- **Stringable Interface**: (8.0+) Classes with `__toString`.

```php
<?php
if (preg_match("/\w+@\w+.\w+/", "user@example.com")) {
    echo "Valid email";
}
```

Edge Cases:

- **UTF-8 Encoding**: Non-UTF-8 strings cause issues; use `mb_string`.
- **Heredoc Indentation**: (7.3+) Flexible closing markers.
- **String Offsets**: `$str[0] = "x"` deprecated in 8.0; use `substr_replace`.

Cross-Connections:

- **Arrays**: Strings split into arrays.
- **Web Development**: Strings in templates and JSON.
- **Security**: Sanitize strings for SQL/HTML.

Objects

Objects are instances of classes or `stdClass` for dynamic properties.

```php
<?php
$obj = new stdClass();
$obj->prop = "value";
```

```
print_r($obj); // stdClass Object ( [prop] => value )

// Array to object
$arr = ["a" => 1];
$obj = (object)$arr;
echo $obj->a; // 1
```

Advanced Features:

- **Serialization**: `serialize`, `unserialize`, `__serialize` (7.4+).
- **Cloning**: `__clone` for deep copies.
- **Magic Methods**: `__get`, `__set`, `__call`.

Edge Cases:

- **Property Overwrites**: Dynamic properties on typed classes (8.0+) restricted.
- **Serialization Security**: `unserialize` can execute `__wakeup`; use `allowed_classes`.
- **Object Comparison**: `==` checks properties; `===` checks identity.

Cross-Connections:

- **OOP**: Objects are central to classes.
- **Arrays**: Objects and arrays interconvert.
- **Database**: Objects represent rows in ORMs.

Object-Oriented Programming

Classes and Objects

Classes define blueprints with properties and methods.

```
<?php
class Person {
    public string $name;
    private int $age;

    public function __construct(string $name, int $age) {
        $this->name = $name;
        $this->age = $age;
    }

    public function introduce(): string {
        return "I'm {$this->name}, {$this->age} years old";
    }

    public static function isAdult(int $age): bool {
        return $age >= 18;
    }
}

$p = new Person("Alice", 30);
```

```
echo $p->introduce(); // I'm Alice, 30 years old
echo Person::isAdult(20); // true
```

Advanced Features:

- **Typed Properties**: (7.4+) `public int $age`.
- **Readonly Properties**: (8.1+) `public readonly int $age`.
- **Constructor Property Promotion**: (8.0+) `public function __construct(public string $name)`.

```
<?php
class Circle {
    public function __construct(public readonly float $radius) {}

    public function area(): float {
        return pi() * $this->radius ** 2;
    }
}

$c = new Circle(5);
echo $c->area(); // 78.539816339745
```

Edge Cases:

- **Property Visibility**: Private properties inaccessible outside class.
- **Readonly Mutation**: Attempting to modify readonly properties throws `Error`.
- **Object Destruction**: `__destruct` may not run in shutdown sequences.

Cross-Connections:

- **Traits**: Reuse code across classes.
- **Namespaces**: Organize classes.
- **Web Frameworks**: Classes power controllers/models.

Inheritance and Interfaces

Inheritance extends classes; interfaces define contracts.

```
<?php
interface Animal {
    public function speak(): string;
}

abstract class BaseAnimal {
    protected string $name;

    public function __construct(string $name) {
        $this->name = $name;
    }
}

class Dog extends BaseAnimal implements Animal {
```

```
    public function speak(): string {
        return "{$this->name} says Woof!";
    }
}

$dog = new Dog("Buddy");
echo $dog->speak(); // Buddy says Woof!
```

Advanced Features:

- **Final Classes/Methods**: Prevent extension/overriding.
- **Anonymous Classes**: Inline class definitions (7.0+).
- **Enum Types**: (8.1+) `enum Status: string { case Active = 'active'; }`.

```
<?php
enum Status: string {
    case Active = 'active';
    case Inactive = 'inactive';
}

function checkStatus(Status $status): string {
    return $status->value;
}

echo checkStatus(Status::Active); // active
```

Edge Cases:

- **Multiple Inheritance**: Not supported; use traits.
- **Interface Conflicts**: Implementing multiple interfaces with same method signatures.
- **Enum Exhaustiveness**: `match` with enums ensures all cases handled.

Cross-Connections:

- **Traits**: Complement inheritance.
- **Type Hints**: Interfaces/enums enhance type safety.
- **Testing**: Mock interfaces in tests.

Traits

Traits provide horizontal code reuse.

```
<?php
trait Loggable {
    public function log(string $message): void {
        echo "Log: $message\n";
    }
}

trait Timestampable {
    public function getTimestamp(): string {
        return date("Y-m-d H:i:s");
```

```
    }
}

class Document {
    use Loggable, Timestampable;

    public function save(): void {
        $this->log("Saving at {$this->getTimestamp()}");
    }
}

$doc = new Document();
$doc->save(); // Log: Saving at 2025-06-17 10:25:00
```

Advanced Features:

- **Trait Precedence**: Overrides inherited methods.
- **Conflict Resolution**: `insteadof` and `as` keywords.
- **Abstract Methods**: Traits can declare abstract methods.

```
<?php
trait A {
    public function method() { echo "A"; }
}

trait B {
    public function method() { echo "B"; }
}

class C {
    use A, B {
        B::method insteadof A;
        A::method as methodA;
    }
}

$c = new C();
$c->method(); // B
$c->methodA(); // A
```

Edge Cases:

- **Conflict Ambiguity**: Unresolved conflicts throw fatal errors.
- **Property Conflicts**: Traits with properties cause issues pre-8.0.
- **Serialization**: Traits don’t affect serialization directly.

Cross-Connections:

- **OOP**: Traits enhance class design.
- **Web Frameworks**: Traits in Laravel/Symfony for utilities.
- **Testing**: Mock trait methods.

Namespaces and Autoloading

Namespaces organize code; autoloading loads classes dynamically.

```
<?php
// src/App/Models/User.php
namespace App\Models;

class User {
    public function __construct(public string $name) {}
}
```

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

use App\Models\User;

$user = new User("Alice");
echo $user->name; // Alice
```

Composer Autoloading (`composer.json`):

```
{
    "autoload": {
        "psr-4": {
            "App": "src/App/"
        }
    }
}
```

Run:

```
composer dump-autoload
```

Advanced Features:

- **PSR-4/PSR-0**: Standard autoloading formats.
- **Alias Imports**: `use App\Models\User as AppUser`.
- **Dynamic Resolution**: `class_exists` with autoloading.

Edge Cases:

- **Case Sensitivity**: Filesystems may cause issues on case-insensitive OS.
- **Autoload Order**: Multiple autoloaders can conflict.
- **Performance**: Optimize with `composer dump-autoload -o`.

Cross-Connections:

- **OOP**: Namespaces organize classes.
- **Web Frameworks**: Frameworks rely on autoloading.
- **Packaging**: Composer manages dependencies.

Error Handling

Exceptions

PHP uses `try`/`catch` for exceptions, with `throw` to raise them.

```
<?php
function divide(float $a, float $b): float {
    if ($b === 0.0) {
        throw new InvalidArgumentException("Division by zero");
    }
    return $a / $b;
}

try {
    echo divide(10, 0);
} catch (InvalidArgumentException $e) {
    echo "Error: {$e->getMessage()}";
} finally {
    echo "\nCleanup";
}
```

Advanced Features:

- **Exception Hierarchy**: Extends `Throwable` (`Error`, `Exception`).
- **Custom Exceptions**: Subclass `Exception`.
- **Multi-Catch**: (7.1+) `catch (Type1|Type2 $e)`.

```
<?php
class CustomException extends Exception {
    public function __construct(string $message, public int $code = 0) {
        parent::__construct($message, $code);
    }
}

try {
    throw new CustomException("Failed", 400);
} catch (CustomException $e) {
    echo "Error: {$e->getMessage()}, Code: {$e->code}";
}
```

Edge Cases:

- **Uncaught Exceptions**: Terminate script; use `set_exception_handler`.
- **Error vs. Exception**: `Error` for internal issues (e.g., type errors).
- **Finally Precedence**: Runs even with `return`.

Cross-Connections:

- **OOP**: Exceptions are classes.
- **Web Development**: Handle exceptions in controllers.
- **Testing**: Assert exceptions in tests.

Error Levels

PHP errors (e.g., notices, warnings) differ from exceptions.

```
<?php
set_error_handler(function (int $severity, string $message, string $file, int $line)
{
    throw new ErrorException($message, 0, $severity, $file, $line);
});

try {
    $x = $undefined; // Notice
} catch (ErrorException $e) {
    echo "Error: {$e->getMessage()}";
}
```

Advanced Features:

- **Error Reporting**: `error_reporting(E_ALL)` for debugging.
- **Display Errors**: `ini_set('display_errors', 1)` in development.
- **Error to Exception**: `ErrorException` bridges errors and exceptions.

Edge Cases:

- **Fatal Errors**: Not catchable pre-7.0; use shutdown handlers.
- **Error Suppression**: `@` operator hides errors but is discouraged.
- **Performance**: Error handling adds overhead in tight loops.

Cross-Connections:

- **Exceptions**: Errors can be converted to exceptions.
- **Web Frameworks**: Frameworks handle errors centrally.
- **Security**: Hide errors in production to avoid leaks.

Functional Programming

Closures and Anonymous Functions

Closures capture variables; arrow functions (7.4+) are concise.

```
<?php
$greeting = "Hello";
$closure = function (string $name) use ($greeting): string {
    return "$greeting, $name!";
};

$arrow = fn(string $name): string => "Hi, $name!";

echo $closure("Alice"); // Hello, Alice!
echo $arrow("Bob"); // Hi, Bob!

// Higher-order function
```

```
function apply(callable $fn, $value) {
    return $fn($value);
}

echo apply(fn($x): int => $x * 2, 5); // 10
```

Advanced Features:

- **Use vs. By-Reference**: `use (&$var)` for mutable captures.
- **First-Class Callables**: (8.1+) `$fn = strlen(...)`.
- **Partial Application**: Manual currying with closures.

```
<?php
function curry(callable $fn, $arg1) {
    return fn(...$args) => $fn($arg1, ...$args);
}

$addFive = curry(fn($a, $b) => $a + $b, 5);
echo $addFive(3); // 8
```

Edge Cases:

- **Closure Scope**: Captured variables persist until closure is unset.
- **Serialization**: Closures are not serializable without libraries.
- **Performance**: Closures slightly slower than named functions.

Cross-Connections:

- **Arrays**: Array functions use callables.
- **OOP**: Closures in methods capture `$this`.
- **Web Development**: Closures in route handlers.

Array Functions

PHP provides functional-style array operations.

```
<?php
$numbers = [1, 2, 3, 4];
$evens = array_filter($numbers, fn($x) => $x % 2 === 0);
$squares = array_map(fn($x) => $x ** 2, $numbers);
$sum = array_reduce($numbers, fn($carry, $x) => $carry + $x, 0);

print_r($evens); // [2, 4]
print_r($squares); // [1, 4, 9, 16]
echo $sum; // 10
```

Advanced Features:

- **array_walk**: Mutates arrays with callbacks.
- **array_column**: Extracts column from array of arrays/objects.
- **array_merge_recursive**: Deep merging of arrays.

Edge Cases:

- **Key Preservation**: `array_map` preserves keys; `array_filter` reindexes numeric keys.
- **Reference Issues**: Callbacks modifying arrays can cause bugs.
- **Memory Usage**: Large arrays with `array_map` create copies.

Cross-Connections:

- **Closures**: Callbacks are often closures.
- **Web Development**: Process form or JSON data.
- **Testing**: Test array transformations.

Web Development Basics

Handling HTTP Requests

PHP handles GET/POST requests via superglobals.

```
<?php
// index.php
$name = $_GET['name'] ?? 'Guest';
echo "Hello, $name!";

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $data = $_POST['data'] ?? '';
    echo "Received: $data";
}
```

Advanced Features:

- **Query Parsing**: `parse_url`, `parse_str`.
- **Request Headers**: `getallheaders()` or `$_SERVER`.
- **RESTful Routing**: Manual routing or frameworks.

Edge Cases:

- **Superglobal Injection**: Sanitize `$_GET`/`_POST` to prevent attacks.
- **Max Input Vars**: `php.ini` limits form fields.
- **URL Encoding**: Use `urlencode`/`urldecode`.

Cross-Connections:

- **Security**: Validate/sanitize inputs.
- **Frameworks**: Replace manual handling.
- **Sessions**: Store request state.

Forms and File Uploads

Handle form submissions and files via `$_POST` and `$_FILES`.

```
<!-- form.html -->
<form method="POST" action="upload.php" enctype="multipart/form-data">
    <input type="text" name="username">
    <input type="file" name="avatar">
    <button type="submit">Submit</button>
</form>
```

```
<?php
// upload.php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
    $file = $_FILES['avatar'];

    if ($file['error'] === UPLOAD_ERR_OK) {
        move_uploaded_file($file['tmp_name'], "uploads/{$file['name']}");
        echo "Uploaded for $username";
    } else {
        echo "Upload failed";
    }
}
```

Advanced Features:

- **Input Filtering**: `filter_input` for sanitization.
- **File Validation**: Check MIME types, size limits.
- **CSRF Protection**: Use tokens to prevent cross-site request forgery.

Edge Cases:

- **File Size Limits**: `php.ini` settings (`upload_max_filesize`, `post_max_size`).
- **Temporary Files**: `$_FILES['tmp_name']` deleted after request.
- **CSRF Attacks**: Always validate tokens in POST forms.

Cross-Connections:

- **Security**: Prevent file injection attacks.
- **Database**: Store file metadata.
- **Frameworks**: Simplify form handling.

Sessions and Cookies

Manage state with sessions and cookies.

```
<?php
// login.php
session_start();

$_SESSION['user'] = 'Alice';
setcookie('theme', 'dark', time() + 3600);

echo "Logged in as {$_SESSION['user']}";
```

```php
<?php
// profile.php
session_start();

if (isset($_SESSION['user'])) {
    $theme = $_COOKIE['theme'] ?? 'light';
    echo "Welcome, {$_SESSION['user']}! Theme: $theme";
} else {
    echo "Please log in";
}
```

Advanced Features:

- **Session Configuration**: `session.save_path`, `session.gc_maxlifetime`.
- **Secure Cookies**: `httponly`, `secure`, `samesite` attributes.
- **Session Storage**: Use Redis/Memcached for scalability.

Edge Cases:

- **Session Hijacking**: Use HTTPS, regenerate session IDs.
- **Cookie Limits**: Browsers cap cookie size/number.
- **Session Lock**: Concurrent requests may block; use `session_write_close`.

Cross-Connections:

- **Security**: Protect session data.
- **Web Frameworks**: Built-in session management.
- **Database**: Store session data for persistence.

Database Integration

PDO

PDO provides a database-agnostic interface.

```php
<?php
try {
    $pdo = new PDO("mysql:host=localhost;dbname=test", "user", "pass");
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
    $stmt->execute(['id' => 1]);
    $user = $stmt->fetch(PDO::FETCH_ASSOC);

    print_r($user);
} catch (PDOException $e) {
    echo "Error: {$e->getMessage()}";
}
```

Advanced Features:

- **Prepared Statements**: Prevent SQL injection.

- **Transactions**: `$pdo->beginTransaction()`, `commit()`, `rollBack()`.
- **Fetch Modes**: `FETCH_OBJ`, `FETCH_CLASS` for objects.

Edge Cases:

- **Connection Errors**: Handle `PDOException` for failed connections.
- **Driver Support**: Not all databases support all PDO features.
- **Persistent Connections**: Risk resource leaks if mismanaged.

Cross-Connections:

- **Security**: PDO prevents injection.
- **ORMs**: Built on PDO.
- **Web Development**: Query data for APIs.

MySQLi

MySQLi is MySQL-specific, offering procedural and OOP interfaces.

```
<?php
$mysqli = new mysqli("localhost", "user", "pass", "test");

if ($mysqli->connect_error) {
    die("Connection failed: {$mysqli->connect_error}");
}

$stmt = $mysqli->prepare("INSERT INTO users (name) VALUES (?)");
$stmt->bind_param("s", $name);
$name = "Alice";
$stmt->execute();
$stmt->close();

$result = $mysqli->query("SELECT * FROM users");
while ($row = $result->fetch_assoc()) {
    print_r($row);
}

$mysqli->close();
```

Advanced Features:

- **Multi-Query**: Execute multiple SQL statements.
- **Asynchronous Queries**: `mysqli_poll` for non-blocking.
- **SSL Support**: Secure database connections.

Edge Cases:

- **SQL Injection**: Always use prepared statements.
- **Resource Management**: Close statements/connections.

- **Character Encoding**: Set `charset` to avoid UTF-8 issues.

Cross-Connections:

- **PDO**: Alternative database interface.
- **Web Frameworks**: Integrate with MySQLi.
- **Security**: Sanitize inputs.

ORMs

Object-Relational Mappers like Doctrine or Eloquent simplify database operations.

```
<?php
// Using Laravel's Eloquent (via Composer)
require 'vendor/autoload.php';

use Illuminate\Database\Capsule\Manager as DB;

$capsule = new DB;
$capsule->addConnection([
    'driver' => 'mysql',
    'host' => 'localhost',
    'database' => 'test',
    'username' => 'user',
    'password' => 'pass',
]);
$capsule->setAsGlobal();
$capsule->bootEloquent();

class User extends Illuminate\Database\Eloquent\Model {}

$user = User::create(['name' => 'Alice']);
$users = User::where('name', 'Alice')->get();
print_r($users->toArray());
```

Advanced Features:

- **Relationships**: Define one-to-many, many-to-many.
- **Query Builder**: Fluent SQL generation.
- **Migration Tools**: Schema management.

Edge Cases:

- **N+1 Problem**: Eager load relations to avoid query overhead.
- **Performance**: ORMs may generate suboptimal SQL.
- **Transaction Scope**: Ensure transactions wrap related operations.

Cross-Connections:

- **Web Frameworks**: Eloquent in Laravel, Doctrine in Symfony.
- **OOP**: Models are classes.

- **Testing**: Mock database queries.

Advanced Topics

Attributes

Attributes (8.0+) provide metadata for classes, methods, and properties.

```
<?php
#[Route("/greet")]
class Controller {
    #[Inject]
    public Service $service;

    #[Get]
    public function greet(): string {
        return $this->service->greet();
    }
}

$ref = new ReflectionClass(Controller::class);
$attrs = $ref->getAttributes();
print_r($attrs); // Array of ReflectionAttribute
```

Advanced Features:

- **Custom Attributes**: Define with `#[Attribute]`.
- **Reflection API**: Access attributes at runtime.
- **Framework Integration**: Laravel/Symfony use attributes for routing.

Edge Cases:

- **Performance**: Reflection is slower than direct access.
- **Attribute Scope**: Limited to classes, methods, properties, etc.
- **Validation**: Ensure attribute arguments are valid.

Cross-Connections:

- **OOP**: Attributes enhance class metadata.
- **Web Frameworks**: Replace annotations.
- **Testing**: Mock attribute-based behavior.

Generators

Generators yield values lazily, saving memory.

```
<?php
function fibonacci(int $n): Generator {
    $a = 0;
    $b = 1;
    for ($i = 0; $i < $n; $i++) {
        yield $a;
```

```
        [$a, $b] = [$b, $a + $b];
    }
}

foreach (fibonacci(5) as $value) {
    echo "$value, "; // 0, 1, 1, 2, 3,
}
```

Advanced Features:

- **Yield From**: Delegate to another generator.
- **Return Values**: Generators can `return` (7.0+).
- **Send**: Pass values back to generator.

Edge Cases:

- **Generator Exhaustion**: Iterating twice requires rewinding.
- **Memory Leaks**: Generators hold state until unset.
- **Invalid Yields**: Non-iterable contexts throw errors.

Cross-Connections:

- **Functional Programming**: Generators enable lazy evaluation.
- **Database**: Stream large result sets.
- **Web Development**: Process large datasets in APIs.

Weak References

Weak references (7.4+) allow non-owning references to objects.

```
<?php
$obj = new stdClass();
$weak = WeakReference::create($obj);
var_dump($weak->get()); // object(stdClass)

unset($obj);
var_dump($weak->get()); // null
```

Advanced Features:

- **WeakMap**: (8.0+) Key-value store with weak references.
- **Caching**: Use weak references for memory-efficient caches.
- **Event Listeners**: Prevent memory leaks in observers.

Edge Cases:

- **Garbage Collection**: Weak references rely on GC cycles.
- **WeakMap Keys**: Only objects allowed as keys.
- **Serialization**: Weak references not serializable.

Cross-Connections:

- **OOP**: Manage object lifecycles.
- **Memory Management**: Prevent leaks.
- **Web Frameworks**: Cache objects in long-running apps.

FFI (Foreign Function Interface)

FFI (7.4+) calls C libraries directly.

```
<?php
$ffi = FFI::cdef("int double(int x);", "./libmyext.so");
echo $ffi->double(5); // 10
```

C Code (`libmyext.c`):

```
int double(int x) {
    return x * 2;
}
```

Compile:

```
gcc -shared -o libmyext.so libmyext.c
```

Advanced Features:

- **Structs**: Define C structs in PHP.
- **Callbacks**: Pass PHP closures to C.
- **Preloading**: Load FFI at startup for performance.

Edge Cases:

- **Platform Dependency**: Requires compiled libraries.
- **Memory Safety**: Incorrect pointer use causes crashes.
- **GIL Absence**: PHP's threading model limits FFI parallelism.

Cross-Connections:

- **Performance**: Optimize bottlenecks.
- **Web Development**: Integrate with C libraries.
- **Packaging**: Include shared libraries.

Performance Optimization

Optimize PHP with profiling and tools.

```
<?php
// Profile with xdebug
function slow() {
    $sum = 0;
    for ($i = 0; $i < 1000000; $i++) {
        $sum += $i;
```

```
    }
    return $sum;
}

echo slow();
```

Tools:

- **OPcache**: Bytecode caching (`opcache.enable=1` in `php.ini`).
- **Xdebug**: Profiling and debugging.
- **PHP JIT**: (8.0+) Enable with `opcache.jit=1205`.

Advanced Features:

- **Blackfire**: SaaS profiling tool.
- **Swoole**: Async PHP for high-performance servers.
- **Static Analysis**: `phpstan`, `psalm` for optimization hints.

Edge Cases:

- **JIT Limitations**: Not all code benefits from JIT.
- **OPcache Invalidation**: Clear cache after code changes.
- **Profiling Overhead**: Slows execution during profiling.

Cross-Connections:

- **FFI**: Optimize with C.
- **Web Frameworks**: Optimize routes/queries.
- **Database**: Index queries for speed.

Asynchronous PHP

Async PHP uses extensions like Swoole or ReactPHP.

```
<?php
// Using ReactPHP (via Composer)
require 'vendor/autoload.php';

use React\EventLoop\Loop;
use React\Http\HttpServer;
use Psr\Http\Message\ServerRequestInterface;

$server = new HttpServer(function (ServerRequestInterface $request) {
    return new Response(200, [], "Hello, Async!");
});

$socket = new React\Socket\ServerSocket('0.0.0.0:8080', Loop::get());
$server->listen($socket);

Loop::run();
```

Advanced Features:

- **Promises**: ReactPHP's promise-based async.
- **Coroutines**: Swoole's fiber-like concurrency (8.1+ fibers enhance).
- **Event Loops**: Non-blocking I/O.

Edge Cases:

- **Blocking Code**: Avoid in async loops; use async libraries.
- **Extension Dependency**: Requires non-standard extensions.
- **Scalability**: Async benefits high-concurrency apps.

Cross-Connections:

- **Web Frameworks**: Laravel Octane uses Swoole.
- **Database**: Async queries with async drivers.
- **Performance**: Async improves throughput.

Security Best Practices

Secure PHP applications against common vulnerabilities.

```
<?php
// SQL injection prevention
$stmt = $pdo->prepare("SELECT * FROM users WHERE name = ?");
$stmt->execute([$_POST['name']]);

// XSS prevention
echo htmlspecialchars($_POST['comment'], ENT_QUOTES, 'UTF-8');

// CSRF token
session_start();
$token = bin2hex(random_bytes(32));
$_SESSION['csrf_token'] = $token;

<input type="hidden" name="csrf_token" value="<?php echo htmlspecialchars($token);
?>">
```

Advanced Practices:

- **Password Hashing**: `password_hash`, `password_verify`.
- **Cryptography**: Use `sodium` for encryption.
- **Secure Headers**: `X-Frame-Options`, `Content-Security-Policy`.

Edge Cases:

- **Input Validation**: Always validate, never trust user input.
- **File Uploads**: Restrict file types, scan for malware.
- **Error Leaks**: Disable `display_errors` in production.

Cross-Connections:

- **Web Development**: Secure forms and APIs.
- **Database**: Prevent injection.
- **Frameworks**: Built-in security features.

PHP Internals

Understand PHP's C-based engine (Zend Engine).

- **Zend Engine**: Interprets and executes PHP code.
- **Opcodes**: Bytecode cached by OPcache.
- **Memory Management**: Reference counting with cycle detection.

```
<?php
// Debug internals
echo "Memory: " . memory_get_usage() . "\n";
$x = [];
$x[] = &$x; // Cycle
unset($x);
gc_collect_cycles();
echo "Memory: " . memory_get_usage();
```

Advanced Features:

- **Extension Development**: Write C extensions for PHP.
- **Zval**: Internal type/value structure for variables.
- **Profiling**: Analyze opcodes with `vld`.

Edge Cases:

- **Memory Leaks**: Cycles require GC.
- **Thread Safety**: PHP not thread-safe by default.
- **Extension Compatibility**: Breaks across PHP versions.

Cross-Connections:

- **FFI**: Interact with C code.
- **Performance**: Optimize based on engine.
- **Web Development**: Extensions enhance frameworks.

Testing in PHP

PHPUnit

PHPUnit is the standard testing framework.

```
<?php
// tests/UnitTest.php
```

```
use PHPUnit\Framework\TestCase;

class MathTest extends TestCase {
    public function testAddition(): void {
        $this->assertEquals(5, add(2, 3));
        $this->assertNotEquals(8, add(2, 3));
    }

    public function testInvalidType(): void {
        $this->expectException(TypeError::class);
        add("2", 3);
    }
}

function add(int $a, int $b): int {
    return $a + $b;
}
```

Run:

```
vendor/bin/phpunit tests
```

Mockery

Mock dependencies with Mockery.

```
<?php
// tests/FeatureTest.php
use Mockery;
use PHPUnit\Framework\TestCase;

class UserServiceTest extends TestCase {
    protected function tearDown(): void {
        Mockery::close();
    }

    public function testFetchUser(): void {
        $repo = Mockery::mock(UserRepository::class);
        $repo->shouldReceive('find')->with(1)->once()->andReturn(['name' =>
'Alice']);

        $service = new UserService($repo);
        $user = $service->getUser(1);

        $this->assertEquals('Alice', $user['name']);
    }
}
```

Code Coverage

Measure coverage with PHPUnit.

```
vendor/bin/phpunit --coverage-html coverage
```

Advanced Features:

- **Data Providers**: `@dataProvider` for parameterized tests.

- **Test Doubles**: Stubs, mocks, spies.
- **Integration Tests**: Test database or API calls.

Edge Cases:

- **Mock Side Effects**: Over-mocking can hide bugs.
- **Coverage Metrics**: High coverage ≠ bug-free.
- **Test Dependencies**: Isolate tests to avoid state leaks.

Cross-Connections:

- **OOP**: Test interfaces and classes.
- **Web Frameworks: Built-in testing tools.
- **Database**: Mock or use test databases.

Web Frameworks

Laravel

Laravel is a full-stack framework with elegant syntax.

```
<?php
// routes/web.php
use App\Http\Controllers\UserController;
use Illuminate\Support\Facades\Route;

Route::get('/greet/{name}', [UserController::class, 'greet']);
```

```
<?php
// app/Http/Controllers/UserController.php
namespace App\Http\Controllers;

class UserController extends Controller {
    public function greet(string $name): array {
        return ['message' => "Hello, $name!"];
    }
}
```

Symfony

Symfony is modular and flexible.

```
<?php
// config/routes.php
use App\Controller\UserController;
use Symfony\Component\Routing\Loader\Configurator\RoutingConfigurator;

return function (RoutingConfigurer $routes) {
    $routes->add('greet', '/greet/{name}')->controller([UserController::class,
'greet']);
};
```

```php
<?php
// src/Controller/UserController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\JsonResponse;

class UserController {
    public function greet(string $name): JsonResponse {
        return new JsonResponse(['message' => "Hello, $name!"]);
    }
}
```

Slim

Slim is a lightweight micro-framework.

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

use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use Slim\Factory\AppFactory;

$app = AppFactory::create();

$app->get('/greet/{name}', function (Request $request, Response $response, array
$args) {
    $name = $args['name'];
    $response->getBody()->write(json_encode(['message' => "Hello, $name!"]));
    return $response->withHeader('Content-Type', 'application/json');
});

$app->run();
```

Advanced Features:

- **Middleware**: Add authentication, logging.
- **Dependency Injection**: Built-in containers.
- **API Development**: REST/GraphQL support.

Edge Cases:

- **Routing Conflicts**: Ensure unique routes.
- **Performance**: Optimize middleware stack.
- **CSRF/XSRF**: Enable protection for forms.

Cross-Connections:

- **Database**: Integrate ORMs.
- **Testing**: Test routes and controllers.
- **Security**: Frameworks enforce best practices.

Building a Sample Project

This project implements a **task management API** using Slim, PDO, async processing (ReactPHP), and FFI, showcasing PHP's modern features.

Project Structure:

```
task-manager/
├── src/
│   ├── App/
│   │   ├── Models/
│   │   │   └── Task.php
│   │   ├── Services/
│   │   │   └── TaskService.php
│   │   └── Routes.php
│   ├── libmyext.so
│   └── libmyext.c
├── tests/
│   ├── Unit/
│   │   └── TaskTest.php
│   └── Feature/
│       └── ApiTest.php
├── public/
│   └── index.php
├── composer.json
└── README.md
```

composer.json:

```
{
    "name": "task-manager",
    "require": {
        "php": "^8.4",
        "slim/slim": "^4.10",
        "slim/psr7": "^1.6",
        "react/http": "^1.9",
        "phpunit/phpunit": "^10.0",
        "mockery/mockery": "^1.5"
    },
    "autoload": {
        "psr-4": {
            "App": "src/App/"
        }
    },
    "require-dev": {
        "phpstan/phpstan": "^1.10"
    }
}
```

libmyext.c:

```
int compute_priority(int id) {
    return id * 3;
}
```

Compile:

```
gcc -shared -o src/libmyext.so src/libmyext.c
```

src/App/Models/Task.php:

```
<?php
namespace App\Models;

enum TaskStatus: string {
    case Pending = 'pending';
    case Completed = 'completed';
    case Failed = 'failed';
}

class Task {
    public function __construct(
        public int $id,
        public string $description,
        public TaskStatus $status = TaskStatus::Pending,
        public int $priority = 0
    ) {}
}
```

src/App/Services/TaskService.php:

```
<?php
namespace App\Services;

use App\Models\Task;
use App\Models\TaskStatus;
use PDO;
use React\Http\HttpServer;
use React\Socket\Server as SocketServer;
use Psr\Http\Message\ServerRequestInterface;
use React\EventLoop\Loop;

class TaskService {
    private PDO $pdo;
    private FFI $ffi;

    public function __construct() {
        $this->pdo = new PDO("sqlite::memory:");
        $this->pdo->exec("CREATE TABLE tasks (id INTEGER PRIMARY KEY, description
TEXT, status TEXT, priority INTEGER)");
        $this->ffi = FFI::cdef("int compute_priority(int id);", __DIR__ .
"/../../libmyext.so");

        // Start async server for task processing
        $this->startAsyncProcessor();
    }

    public function createTask(Task $task): Task {
        $task->priority = $this->ffi->compute_priority($task->id);
        $stmt = $this->pdo->prepare("INSERT INTO tasks (id, description, status,
priority) VALUES (?, ?, ?, ?)");
        $stmt->execute([$task->id, $task->description, $task->status->value, $task-
>priority]);
```

```
        return $task;
    }

    public function getTasks(): array {
        $stmt = $this->pdo->query("SELECT * FROM tasks");
        $tasks = [];
        while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
            $tasks[] = new Task(
                $row['id'],
                $row['description'],
                TaskStatus::from($row['status']),
                $row['priority']
            );
        }
        return $tasks;
    }

    private function startAsyncProcessor(): void {
        $server = new HttpServer(function (ServerRequestInterface $request) {
            $id = (int)($request->getQueryParams()['id'] ?? 0);
            $stmt = $this->pdo->prepare("UPDATE tasks SET status = ? WHERE id = ?");
            $stmt->execute([TaskStatus::Completed->value, $id]);
            return new React\Http\Message\Response(200, [], "Processed task $id");
        });

        $socket = new SocketServer('127.0.0.1:8081', Loop::get());
        $server->listen($socket);
    }
}
```

src/App/Routes.php:

```
<?php
namespace App;

use App\Models\Task;
use App\Models\TaskStatus;
use App\Services\TaskService;
use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use Slim\App;

function registerRoutes(App $app, TaskService $service): void {
    $app->post('/tasks', function (Request $request, Response $response) use
($service): Response {
        $data = $request->getParsedBody();
        $task = new Task(
            (int)($data['id'] ?? 0),
            (string)($data['description'] ?? '')
        );
        $task = $service->createTask($task);
        $response->getBody()->write(json_encode([
            'id' => $task->id,
            'description' => $task->description,
            'status' => $task->status->value,
            'priority' => $task->priority
```

```
        ]));
        return $response->withHeader('Content-Type', 'application/json');
    });

    $app->get('/tasks', function (Request $request, Response $response) use
($service): Response {
        $tasks = $service->getTasks();
        $response->getBody()->write(json_encode(array_map(fn(Task $t) => [
            'id' => $t->id,
            'description' => $t->description,
            'status' => $t->status->value,
            'priority' => $t->priority
        ], $tasks)));
        return $response->withHeader('Content-Type', 'application/json');
    });
}
```

public/index.php:

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

use App\Routes;
use App\Services\TaskService;
use Slim\Factory\AppFactory;

$app = AppFactory::create();
$service = new TaskService();

Routes\registerRoutes($app, $service);

$app->run();
```

tests/Unit/TaskTest.php:

```
<?php
use App\Models\Task;
use App\Models\TaskStatus;
use PHPUnit\Framework\TestCase;

class TaskTest extends TestCase {
    public function testTaskCreation(): void {
        $task = new Task(1, "Test task", TaskStatus::Pending, 10);
        $this->assertEquals(1, $task->id);
        $this->assertEquals("Test task", $task->description);
        $this->assertEquals(TaskStatus::Pending, $task->status);
        $this->assertEquals(10, $task->priority);
    }
}
```

tests/Feature/ApiTest.php:

```
<?php
use App\Services\TaskService;
use PHPUnit\Framework\TestCase;
use Psr\Http\Message\ResponseInterface;
```

```
use Slim\Psr7\Factory\ServerRequestFactory;
use Slim\Psr7\Factory\ResponseFactory;

class ApiTest extends TestCase {
    private TaskService $service;

    protected function setUp(): void {
        $this->service = new TaskService();
    }

    public function testCreateTask(): void {
        $request = (new ServerRequestFactory())->createServerRequest('POST',
'/tasks')
            ->withParsedBody(['id' => 1, 'description' => 'Test']);
        $response = new Slim\Psr7\Response();

        $handler = function ($req, $res) {
            $data = $req->getParsedBody();
            $res->getBody()->write(json_encode(['id' => $data['id'], 'description' =>
$data['description']]));
            return $res->withHeader('Content-Type', 'application/json');
        };

        $response = $handler($request, $response);
        $result = json_decode((string)$response->getBody(), true);

        $this->assertEquals(1, $result['id']);
        $this->assertEquals('Test', $result['description']);
    }
}
```

Features Demonstrated:

- **OOP**: `Task` class, `TaskStatus` enum, `TaskService`.
- **Async PHP**: ReactPHP for async task processing.
- **FFI**: C library for priority computation.
- **Database**: PDO with SQLite in-memory database.
- **Web Development**: Slim framework for REST API.
- **Testing**: PHPUnit for unit and feature tests.
- **Type Safety**: Typed properties, return types, enums.
- **Security**: Input validation in API.
- **Packaging**: Composer for dependencies and autoloading.

Running the Project:

```
cd task-manager
composer install
gcc -shared -o src/libmyext.so src/libmyext.c
php -S localhost:8000 -t public
```

Test:

```
vendor/bin/phpunit tests
```

Sample API Usage:

```
curl -X POST http://localhost:8000/tasks -d '{"id":1,"description":"Do homework"}' -H
"Content-Type: application/json"
curl http://localhost:8000/tasks
```

Edge Cases Handled:

- **Error Handling**: PDO exceptions, JSON validation.
- **Async Safety**: ReactPHP event loop for non-blocking tasks.
- **Security**: Input sanitization, no SQL injection.
- **Type Safety**: Enums and type hints.

Resources

- **Official Docs**: PHP Manual, RFCs.
- **Tutorials**: PHP The Right Way, Laracasts.
- **Community**: Reddit, PHP UG, PHP Chat.
- **Libraries**: Packagist, Awesome PHP.
- **Tools**: PHPStan, Psalm, Composer.
- **Books**: *PHP 8 Objects, Patterns, and Practice* (Apress), *Modern PHP* (O'Reilly).
- **Frameworks**: Laravel Docs, Symfony Docs, Slim Docs.
- **Security**: OWASP PHP Security.

This guide and sample project provide a comprehensive foundation for mastering PHP, from beginner to expert, with practical applications and deep insights into its versatile features.