PHP Interview Questions & Answers

A complete reference covering Core PHP, OOP, PDO/Database, CodeIgniter 3, Security, Error Handling, and Practical Scenarios.

Core PHP Fundamentals

1. What's the difference between `==` and `===`?

`==` compares values after type juggling (converting types to match). `===` compares both value **and** type, strictly.

```php
var_dump(0 == "abc");   // false in PHP 8+ (true in PHP 7 — classic gotcha)
var_dump("1" == 1);     // true  — string "1" converted to int
var_dump("1" === 1);    // false — different types
var_dump(0 == false);   // true
var_dump(0 === false);  // false
```

Rule of thumb: always use `===` unless you specifically want type coercion — it avoids subtle bugs, especially in form/validation logic.

---

2. Difference between `include`, `require`, `include_once`, `require_once`?

	Missing file behavior	Repeated inclusion
`include`	Warning, script continues	Included every time called
`require`	Fatal error, script stops	Included every time called
`include_once`	Warning, script continues	Included only once
`require_once`	Fatal error, script stops	Included only once

```php
require_once 'db_connect.php'; // critical — script can't run without it, and n
include 'optional_widget.php'; // non-critical — page can still render without
```

In CodeIgniter, this is largely abstracted away (`$this->load->model()`), but you'd still see `require_once` in bootstrap/config files.

3. What are PHP superglobals? Name a few.

Superglobals are built-in arrays available in **every scope** (functions, classes, included files) without needing to `global` them.

```php
$_GET       // URL query string params
$_POST      // form-submitted data
$_SERVER    // server/environment info (e.g. $_SERVER['REMOTE_ADDR'])
$_SESSION   // session data
$_COOKIE    // cookies
$_FILES     // uploaded files
$_REQUEST   // merged GET+POST+COOKIE (avoid — ambiguous source)
$GLOBALS    // access to all global-scope variables
```

Example — you've used `$_SERVER['HTTP_AUTHORIZATION']` and `$_SERVER['REMOTE_ADDR']` directly in header/IP-whitelisting code.

---

4. Difference between `isset()`, `empty()`, and `is_null()`?

```php
$a = null;
$b = '';
$c = 0;
$d = 'hello';

isset($a);   // false — isset() is false for null
isset($b);   // true  — variable exists, even though it's an empty string
empty($a);   // true
empty($b);   // true  — empty string counts as empty
empty($c);   // true  — 0 counts as empty!
empty($d);   // false
is_null($a); // true
```

Practical gotcha you've already hit: `empty($amount)` will treat a legitimate `0` (e.g., a discount or tax of zero) as "empty" — this is why for numeric fields you sometimes need `$amount !== '' && $amount !== null` instead of `!empty($amount)`.

---

5. What's the difference between `static` and `self` in PHP?

`self::` refers to the class where the code is **literally written**. `static::` refers to the class that was **actually called** at runtime (late static binding) — relevant when a child class extends a parent.

```php
class Model
{
    public static function create()
    {
        return new self();   // always creates a Model, even if called from a c
    }

    public static function make()
    {
        return new static(); // creates whichever class was actually called
    }
}

class Contract extends Model {}

$a = Contract::create(); // returns a Model instance (wrong!)
$b = Contract::make();   // returns a Contract instance (correct)
```

---

6. Explain passing by value vs. passing by reference (`&$var`).

By default, PHP passes variables **by value** — the function gets a copy. Using `&` passes **by reference** — the function can modify the original.

```php
function addTax($amount) {
    $amount += ($amount * 0.06); // modifies the copy only
}
$price = 1000;
addTax($price);
echo $price; // still 1000

function addTaxRef(&$amount) {
    $amount += ($amount * 0.06); // modifies the original
}
addTaxRef($price);
echo $price; // 1060
```

Objects are a special case — they're passed by **handle** (a reference to the object), so modifying an object's properties inside a function affects the original object even without `&`.

---

7. What are magic methods? Name a few.

Special methods PHP calls automatically in certain situations, always prefixed with `__`.

```php
class Contract
{
    private $data = [];

    public function __construct($id)      // called automatically on `new Contr
    {
        $this->data['id'] = $id;
    }

    public function __get($name)           // called when accessing an undefine
    {
        return $this->data[$name] ?? null;
    }

    public function __set($name, $value)   // called when setting an undefined/
    {
        $this->data[$name] = $value;
    }

    public function __toString()           // called when the object is treated
    {
        return "Contract #{$this->data['id']}";
    }

    public function __call($name, $args)   // called when an undefined method i
    {
        return "Method $name doesn't exist";
    }
}

$c = new Contract(101);
$c->status = 'active';   // triggers __set
echo $c->status;         // triggers __get
echo $c;                 // triggers __toString → "Contract #101"
```

8. Difference between session and cookie?

	Session	Cookie
Stored where	Server (file/DB/Redis), only a session ID is sent to the browser	Entirely on the client (browser)
Security	More secure — data never leaves the server	Less secure — visible/editable on client side
Size limit	Effectively unlimited (server storage)	~4KB per cookie
Lifetime	Ends on browser close (by default) or configured timeout	Persists until expiry date you set, even after closing browser

```php
// Session
session_start();
$_SESSION['user_id'] = 55;

// Cookie
setcookie('remember_token', $token, time() + 86400 * 30); // 30 days
```

For login/auth state (e.g. your EPMS login), sessions are the right tool. Cookies are better for things like "remember me" tokens or non-sensitive UI preferences.

---

9. What is type juggling / type coercion in PHP?

PHP automatically converts a value's type depending on context, since it's loosely typed.

```php
$result = "5" + 3;         // 8 (int) — string numeric auto-converted
$result = "5" . 3;         // "53" (string) — concatenation forces string contex
$result = "5apples" + 3;   // TypeError in PHP 8 (used to silently give 8 with a
if ("0") { ... }           // false — "0" is falsy
if ("0.0") { ... }         // true  — "0.0" is truthy! (classic gotcha)
```

This is exactly why `===` and explicit casting (`(int)`, `(float)`) matter in validation logic — especially for financial fields like your withholding tax calculations.

---

10. Difference between `array_map`, `array_filter`, `array_walk`?

```php
$prices = [100, 200, 300];

// array_map — transforms every element, returns a NEW array
$withTax = array_map(fn($p) => $p * 1.06, $prices);
// [106, 212, 318]

// array_filter — keeps elements matching a condition, returns a NEW array (key
$expensive = array_filter($prices, fn($p) => $p > 150);
// [1 => 200, 2 => 300]

// array_walk — modifies the array IN PLACE via reference, returns bool
array_walk($prices, function (&$p) { $p = $p * 1.06; });
// $prices is now [106, 212, 318]
```

OOP

11. Explain the 4 pillars of OOP with examples.

Encapsulation — bundle data + methods, hide internals behind `private`/`protected`.

```php
class Contract {
    private $advertiseDate;
    public function setAdvertiseDate($d) { $this->advertiseDate = $d; }
}
```

Abstraction — expose only the "what," hide the "how."

```php
abstract class BaseModel {
    abstract public function save(); // forces implementation, hides HOW it's d
}
```

Inheritance — reuse code via parent-child relationships.

```php
class Model { protected $pdo; }
class OrganizationModel extends Model { /* inherits $pdo */ }
```

Polymorphism — same method call, different behavior depending on the object.

```php
interface ReportExporter { public function export(); }
class TenderReportExcel implements ReportExporter { public function export() {
class OrganizationSummaryReportExcel implements ReportExporter { public functio

function generate(ReportExporter $r) { $r->export(); } // works with either cla
```

---

12. Difference between abstract class and interface — when to use which?

	Abstract class	Interface
Method bodies	Can have some implemented, some abstract	None — only signatures
Properties	Can have real properties	Only constants
Inheritance	Single (`extends` one class only)	Multiple (`implements` many)
Use case	"Is-a" relationship with shared base logic	"Can-do" contract across unrelated classes

```php
abstract class BaseModel {
    protected $pdo;
    public function __construct($pdo) { $this->pdo = $pdo; } // shared logic
    abstract public function save();                          // must be implem
}

interface Cacheable {
    public function getCacheKey();
}

class Product extends BaseModel implements Cacheable {
    public function save() { /* ... */ }
    public function getCacheKey() { return "product_{$this->id}"; }
}
```

Use an **abstract class** when subclasses share real, reusable logic (like your PDO singleton base model). Use an **interface** when unrelated classes just need to guarantee the same method exists (like your two Excel report classes both needing `export()`).

13. What is a trait, and why does PHP need it?

PHP only allows single inheritance (`extends` one class), but sometimes unrelated classes need the **same reusable code**. Traits solve this by injecting method bodies into a class without a parent-child relationship.

```php
trait DateFormatterTrait {
    protected function formatDate($date) {
        return !empty($date) ? date('Y-m-d', strtotime($date)) : null;
    }
}

class Contract {
    use DateFormatterTrait;
}
class Addendum {
    use DateFormatterTrait; // same logic, unrelated class
}
```

---

14. What's the difference between `public`, `protected`, `private`?

```php
class Model {
    public $name;       // accessible from anywhere
    protected $pdo;      // accessible in this class AND subclasses
    private $apiSecret;  // accessible ONLY inside this exact class
}

class OrganizationModel extends Model {
    public function test() {
        echo $this->pdo;       // OK — protected, inherited
        echo $this->apiSecret; // ERROR — private, not visible to subclass
    }
}
```

---

15. What is late static binding (`static::` vs `self::`)?

Covered in Q5 above — `static::` resolves to the calling class at runtime, `self::` always resolves to the class where it's written, regardless of who calls it.

16. Explain dependency injection.

Instead of a class creating its own dependencies internally (tight coupling), you "inject" them from outside — usually via the constructor. This makes classes easier to test and swap.

```php
// Without DI — tightly coupled, hard to test
class OrganizationModel {
    private $pdo;
    public function __construct() {
        $this->pdo = new PDO('mysql:host=localhost;dbname=epms', 'user', 'pass'
    }
}

// With DI — the dependency is passed in
class OrganizationModel {
    private $pdo;
    public function __construct(PDO $pdo) {
        $this->pdo = $pdo; // could be a real connection OR a mock for testing
    }
}

$pdo = new PDO(/* ... */);
$model = new OrganizationModel($pdo);
```

---

17. What's a singleton pattern?

Ensures a class has **only one instance** throughout the app's lifecycle, with a global access point — exactly what you built for your SwiftPOS PDO connection, avoiding creating a new DB connection every time.

```php
class Database {
    private static ?Database $instance = null;
    private PDO $pdo;

    private function __construct() { // private — can't be `new`'d from outside
        $this->pdo = new PDO('mysql:host=localhost;dbname=swiftpos', 'user', 'p
    }

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

    public function getConnection(): PDO {
        return $this->pdo;
    }
}

$db1 = Database::getInstance();
$db2 = Database::getInstance();
var_dump($db1 === $db2); // true — same instance, same connection
```

---

18. Difference between `instanceof` and `get_class()`?

```php
class Contract {}
class Addendum extends Contract {}

$a = new Addendum();

echo get_class($a);         // "Addendum" — exact class name only
var_dump($a instanceof Contract); // true — checks class OR any parent/interfac
var_dump($a instanceof Addendum); // true
```

`instanceof` is generally preferred for logic checks since it correctly handles inheritance; `get_class()` is more for logging/debugging exact type names.

---

19. Can a final class be extended? Can a final method be overridden?

No to both — `final` locks it down.

```php
final class Config {
    // cannot be extended by any other class
}

class Model {
    final public function connect() {
        // cannot be overridden by any subclass
    }
}
```

Useful when you want to guarantee core behavior (like a security-critical method) can never be silently changed by a subclass.

---

20. Difference between composition and inheritance?

Inheritance — "is-a" relationship.

```php
class Vehicle {}
class Car extends Vehicle {} // a Car IS-A Vehicle
```

Composition — "has-a" relationship, building complex objects out of smaller ones.

```php
class Engine {
    public function start() { return "Engine started"; }
}

class Car {
    private Engine $engine; // Car HAS-A Engine
    public function __construct(Engine $engine) {
        $this->engine = $engine;
    }
    public function drive() {
        return $this->engine->start() . " - driving";
    }
}
```

Modern best practice generally favors **composition over inheritance** — it's more flexible and avoids deep, fragile class hierarchies.

Database / PDO

21. Why use PDO over `mysqli`?

- **Database-agnostic** — same API works with MySQL, PostgreSQL, SQLite, etc. (`mysqli` is MySQL-only)
- **Named + positional placeholders** in prepared statements
- **Object-oriented API** throughout, consistent error handling via exceptions
- Cleaner support for transactions across multiple statements

```php
$pdo = new PDO('mysql:host=localhost;dbname=epms', $user, $pass);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

---

22. What is SQL injection, and how does PDO prevent it?

SQL injection happens when user input is directly concatenated into a SQL query, letting an attacker inject their own SQL logic.

```php
// VULNERABLE
$id = $_GET['id']; // e.g. "1 OR 1=1"
$query = "SELECT * FROM users WHERE id = $id"; // returns ALL users!

// SAFE — prepared statement with bound parameter
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id");
$stmt->execute(['id' => $_GET['id']]);
```

PDO prepared statements send the SQL structure and the data **separately** to the database — the input is never interpreted as executable SQL, no matter what's inside it.

---

23. Difference between `bindParam()` and `bindValue()`?

```php
$stmt = $pdo->prepare("SELECT * FROM contracts WHERE status = :status");

$status = 'active';
$stmt->bindParam(':status', $status); // binds by REFERENCE — reads $status's v
$status = 'archived';                 // changes the value before execute!
$stmt->execute();                     // will use 'archived', not 'active'

$stmt->bindValue(':status', 'active'); // binds the VALUE immediately, fixed at
$stmt->execute();                      // always uses 'active'
```

`bindValue()` is generally safer/more predictable unless you specifically need reference behavior (e.g., binding inside a loop with a changing variable).

---

24. Difference between `PDO::FETCH_ASSOC` vs `PDO::FETCH_OBJ`?

```php
$stmt = $pdo->query("SELECT id, name FROM organizations");

$row = $stmt->fetch(PDO::FETCH_ASSOC);
echo $row['name']; // array access

$stmt2 = $pdo->query("SELECT id, name FROM organizations");
$row2 = $stmt2->fetch(PDO::FETCH_OBJ);
echo $row2->name; // object property access
```

---

25. How do you handle transactions in PDO?

Groups multiple queries so they either **all** succeed or **all** roll back — critical for something like archiving a contract that touches multiple tables.

```php
try {
    $pdo->beginTransaction();

    $pdo->prepare("INSERT INTO archived_contracts (...) VALUES (...)")->execute
    $pdo->prepare("DELETE FROM contracts WHERE id = ?")->execute([$contractId])

    $pdo->commit(); // both succeeded — save permanently
} catch (Exception $e) {
    $pdo->rollBack(); // something failed — undo everything
    error_log("Error archiving contract: " . $e->getMessage());
}
```

---

26. What's `SQLSTATE[HY093]` and what causes it?

`HY093: Invalid parameter number` — almost always means a **mismatch between placeholders and bound values** in a prepared statement. Common causes:

```php
// Mismatched placeholder count
$stmt = $pdo->prepare("SELECT * FROM contracts WHERE org_id = ? AND status = ?"
$stmt->execute([$orgId]); // only 1 value provided for 2 placeholders — HY093

// Mixing named and positional placeholders (not allowed)
$stmt = $pdo->prepare("SELECT * FROM contracts WHERE org_id = :org_id AND statu
$stmt->execute([':org_id' => 1, 'active']); // ERROR — pick one style, not both

// Reusing the same named placeholder twice with different array keys expected
```

Fix: count your `?`/`:name` placeholders against your bound array precisely, and never mix named + positional in the same statement.

---

27. Named vs positional placeholders?

```php
// Positional — order-dependent
$stmt = $pdo->prepare("INSERT INTO contracts (org_id, status) VALUES (?, ?)");
$stmt->execute([$orgId, $status]);

// Named — order-independent, more readable, easier with many columns
$stmt = $pdo->prepare("INSERT INTO contracts (org_id, status) VALUES (:org_id,
$stmt->execute(['org_id' => $orgId, 'status' => $status]);
```

Named placeholders are generally preferred for readability, especially with your `filterParams()`-style dynamic query building.

---

28. What happens inserting `''` into a `DATE`/`DECIMAL` column in strict mode?

MySQL/MariaDB strict mode rejects `''` as an invalid value for these types, throwing `SQLSTATE[22007]` (date) or a truncation/type error (decimal) — exactly the error you ran into.

```php
// Fix pattern
$advDate = !empty($contractFormData['advertise_date'])
    ? date('Y-m-d', strtotime($contractFormData['advertise_date']))
    : null; // NULL is valid, '' is not — assuming column allows NULL
```

CodeIgniter 3

29. Explain MVC as implemented in CodeIgniter.

- **Model** — handles data/DB logic (`application/models/`)
- **View** — handles presentation/HTML (`application/views/`)
- **Controller** — handles request flow, ties Model + View together (`application/controllers/`)

```php
// Controller
class Contracts extends CI_Controller {
    public function index() {
        $this->load->model('Contract_model');
        $data['contracts'] = $this->Contract_model->getAll();
        $this->load->view('contracts/index', $data);
    }
}

// Model
class Contract_model extends CI_Model {
    public function getAll() {
        return $this->db->get('contracts')->result();
    }
}
```

---

30. Difference between a CI helper and a library?

- **Helper** — a set of standalone **functions** (not a class), loaded globally, no state.
- **Library** — a **class** with methods, typically holds state/config, instantiated as an object.

```php
$this->load->helper('date');      // gives you plain functions like now(), mdat
echo mdate('%Y-%m-%d', now());

$this->load->library('upload');   // gives you an object: $this->upload
$this->upload->do_upload();
```

---

31. How does `$this->load->model()` work internally?

It reads the model file from `application/models/`, instantiates the class, and attaches it as a property on the CI super-object (`$this`) so it's accessible from controllers/views via `$this->ModelName`.

```php
$this->load->model('Contract_model');
$this->Contract_model->getAll(); // now available as a property
```

---

32. What is CSRF protection, and how does CI implement it?

CSRF (Cross-Site Request Forgery) tricks a logged-in user's browser into submitting a request they didn't intend (e.g., a hidden auto-submitting form on a malicious site using the victim's session cookie).

CI protects against this by generating a random token per session, embedding it in forms, and rejecting any POST request that doesn't include a matching token.

```php
// config.php
$config['csrf_protection'] = TRUE;

// In the view — CI auto-injects this if csrf_protection is on and you use form
echo form_open('contracts/save');
// <input type="hidden" name="csrf_token_name" value="...">
```

You've already debugged CSRF token issues specifically on PHP 8.2+ — a common cause there is session/cookie `SameSite` attribute changes in newer PHP versions breaking token persistence across requests.

---

33. Many-to-many relationship in CI without an ORM?

Use a **pivot/junction table**, and manually JOIN.

```php
// organizations, tenders, and a pivot: organization_tenders (org_id, tender_id

public function getTendersForOrg($orgId) {
    $this->db->select('t.*');
    $this->db->from('tenders t');
    $this->db->join('organization_tenders ot', 'ot.tender_id = t.id');
    $this->db->where('ot.org_id', $orgId);
    return $this->db->get()->result();
}
```

---

34. `$this->db->get()` vs raw query building in CI?

```php
// Query Builder — safer, more readable, auto-escapes values
$this->db->where('status', 'active');
$result = $this->db->get('contracts')->result();

// Raw query — full control, needed for complex CASE WHEN / subqueries
$result = $this->db->query(
    "SELECT status, COUNT(*) as total FROM contracts
     GROUP BY status"
)->result();
```

Your consolidated statistics query (6-8 queries merged into one `CASE WHEN` aggregate) is a great example of when raw query building outperforms Query Builder — some SQL is too complex to express cleanly through the builder API.

Security

35. CSRF vs XSS — differences and prevention?

	CSRF	XSS
What it does	Tricks the browser into sending an unwanted **request** using the victim's session	Injects malicious **script** into a page that other users view
Prevention	CSRF tokens on forms	Escape/sanitize all output (`htmlspecialchars()`), Content-Security-Policy headers

```php
// XSS prevention — always escape output
echo htmlspecialchars($_POST['comment'], ENT_QUOTES, 'UTF-8');

// Vulnerable
echo $_POST['comment']; // if it contains <script>...</script>, it executes for
```

---

36. How do you securely hash passwords? Why not MD5/SHA1?

MD5/SHA1 are **fast** hashing algorithms — great for checksums, terrible for passwords, because that speed lets attackers brute-force billions of guesses per second (especially with GPU/rainbow tables). They also don't have built-in salting.

```php
// Correct — bcrypt-based, includes automatic salting, deliberately slow
$hash = password_hash($password, PASSWORD_DEFAULT);

// Verify
if (password_verify($inputPassword, $hash)) {
    // correct password
}
```

---

37. Difference between `password_hash()` and `crypt()`?

`password_hash()` is a modern, higher-level wrapper **built on top of** `crypt()` — it automatically generates a secure random salt, uses a safe default algorithm (bcrypt), and handles cost factor configuration for you. `crypt()` requires you to manage all of that manually and is easy to misuse.

```php
// Manual crypt() — error-prone, don't do this
$salt = '$2y$10$' . substr(md5(rand()), 0, 22);
$hash = crypt($password, $salt);

// password_hash() — handles salt + cost automatically
$hash = password_hash($password, PASSWORD_BCRYPT, ['cost' => 12]);
```

If you were seeing "unexpected hash behavior" in your login system, a common cause is **re-hashing an already-hashed password** (e.g. hashing on both insert AND update), or comparing with `==` instead of `password_verify()`.

---

38. What is HMAC signing and when would you use it?

HMAC (Hash-based Message Authentication Code) proves a message wasn't tampered with and came from someone who knows a shared secret key — common in payment gateway integrations (exactly what you built).

```php
$secret = 'shared_secret_key';
$payload = json_encode($data);

$signature = hash_hmac('sha256', $payload, $secret);

// Send $payload + $signature to the API
// Receiver recomputes the HMAC with their copy of the secret and compares — if
```

---

39. How do you securely handle a Bearer token from the Authorization header?

```php
function getBearerToken() {
    $headers = getallheaders();
    foreach ($headers as $name => $value) {
        if (strtolower($name) === 'authorization') {
            if (preg_match('/Bearer\s+(.*)$/i', trim($value), $matches)) {
                return $matches[1];
            }
        }
    }
    return null;
}

$token = getBearerToken();
if (!$token) {
    http_response_code(401);
    die(json_encode(['error' => 'Missing token']));
}
```

Always validate the token (signature/expiry check via JWT, or a DB lookup) — never trust its mere presence as proof of authentication.

Error Handling / Debugging

40. `error_reporting()` vs `display_errors` vs logging to a file?

```php
error_reporting(E_ALL); // WHICH errors to report (all levels)

ini_set('display_errors', 0); // whether to SHOW errors on screen (off in produ
ini_set('log_errors', 1);
ini_set('error_log', '/var/log/php_errors.log'); // WHERE to send them instead
```

In production (like your live EPMS server), `display_errors` should always be `0` — showing raw errors to users can leak file paths, DB structure, or credentials. Logging to a file (which is exactly where you're pulling those `[29-Jun-2026 ...]` error entries from) is the correct approach.

---

41. Difference between `Exception` and `Error` in PHP 7+?

Both implement `Throwable`. `Exception` represents expected, recoverable problems your code raises intentionally. `Error` represents internal PHP engine failures (type errors, fatal issues) that historically weren't catchable before PHP 7.

```php
try {
    throw new InvalidArgumentException("Invalid contract ID");
} catch (Exception $e) {
    echo $e->getMessage();
}

try {
    strlen(); // ArgumentCountError (a subclass of Error) — missing required ar
} catch (Error $e) {
    echo $e->getMessage();
}

// Catch both at once
try {
    // risky code
} catch (Throwable $e) {
    error_log($e->getMessage());
}
```

---

42. `try/catch/finally` — when does `finally` matter?

`finally` always runs, whether an exception was thrown or not — useful for cleanup (closing connections, releasing locks).

```php
$pdo->beginTransaction();
try {
    $pdo->prepare("INSERT INTO contracts (...) VALUES (...)")->execute($data);
    $pdo->commit();
} catch (Exception $e) {
    $pdo->rollBack();
    error_log($e->getMessage());
} finally {
    echo "Archive process finished."; // runs no matter what happened above
}
```

Practical / Scenario-Based

43. "Query throws `SQLSTATE[42S22]: Column not found` — debugging process?"

1. Read the exact column name in the error (`Unknown column 'currency'`).
2. Run `DESCRIBE table_name;` (or `SHOW COLUMNS FROM table_name;`) to confirm the actual schema.
3. Check if the column is missing entirely (needs an `ALTER TABLE ... ADD COLUMN`), or if it exists under a different name (typo/renamed column).
4. Check if the INSERT/SELECT query was copy-pasted from a different table that *does* have that column.
5. Fix either the query (remove/rename the reference) or the schema (add the missing column) — whichever matches the intended behavior.

```sql
ALTER TABLE archived_contracts ADD COLUMN currency VARCHAR(3) DEFAULT 'PKR';
```

---

44. Design a schema for a procurement plan tied to a financial year?

sql

```sql
CREATE TABLE procurement_plans (
    id INT AUTO_INCREMENT PRIMARY KEY,
    organization_id INT NOT NULL,
    financial_year VARCHAR(9) NOT NULL,      -- e.g. '2026-2027'
    item_category VARCHAR(255) NOT NULL,
    description TEXT,
    estimated_cost DECIMAL(15,2) NOT NULL,
    procurement_method VARCHAR(100),          -- open bidding, RFQ, direct cont
    tentative_advertise_date DATE NULL,
    funding_source VARCHAR(255),
    status ENUM('draft','submitted','approved') DEFAULT 'draft',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (organization_id) REFERENCES organizations(id)
);

CREATE TABLE tenders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    procurement_plan_id INT NULL,             -- links a tender back to its pla
    -- ... other tender fields
    FOREIGN KEY (procurement_plan_id) REFERENCES procurement_plans(id)
);
```

---

45. Write a function to extract a Bearer token using regex.

php

```php
function extractBearerToken($authorizationHeader) {
    if (preg_match('/Bearer\s+(\S+)/i', trim($authorizationHeader), $matches))
        return $matches[1]; // $matches[0] = full match, $matches[1] = captured
    }
    return null;
}

echo extractBearerToken('Bearer abc123xyz'); // "abc123xyz"
```

---

46. FIFO costing logic for same-SKU items bought at different costs.

php

```php
class FifoCosting
{
    /**
     * $batches = [
     *   ['qty' => 50, 'cost' => 100],
     *   ['qty' => 30, 'cost' => 120],
     * ]
     * Sells oldest stock first.
     */
    public function calculateCost(array &$batches, int $qtySold): float
    {
        $totalCost = 0;
        $remaining = $qtySold;

        foreach ($batches as &$batch) {
            if ($remaining <= 0) break;

            $take = min($batch['qty'], $remaining);
            $totalCost += $take * $batch['cost'];
            $batch['qty'] -= $take;
            $remaining -= $take;
        }

        // remove fully depleted batches
        $batches = array_filter($batches, fn($b) => $b['qty'] > 0);

        return $totalCost;
    }
}

$batches = [
    ['qty' => 50, 'cost' => 100],
    ['qty' => 30, 'cost' => 120],
];

$fifo = new FifoCosting();
$cost = $fifo->calculateCost($batches, 60);
// Uses all 50 @ 100 = 5000, then 10 @ 120 = 1200 → total 6200
```

End of reference document.