PHP and MySQL Foundations

Connect PHP securely to MySQL and implement validated create, read, filter and update operations with PDO.

Connecting PHP to MySQL

A database connection allows PHP to send SQL to MySQL and receive results. Keep the connection in one reusable file.

Create the connection file

Create `includes/db.php`:

```
<?php
$host = "localhost";
$dbname = "campready_db";
$username = "root";
$password = "";

$dsn = "mysql:host=$host;dbname=$dbname;charset=utf8mb4";

try {
    $pdo = new PDO($dsn, $username, $password, [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
    ]);
} catch (PDOException $error) {
    exit("Database connection unavailable.");
}
```

XAMPP often uses an empty local root password, but never assume this on a hosted server. Do not reveal connection details or raw errors to users.

Use the connection

```
<?php
require __DIR__ . "/includes/db.php";
$stmt = $pdo->query("SELECT activity_id, activity_name FROM activities");
```

```
$activities = $stmt->fetchAll();
```

Why these choices matter

- `utf8mb4` supports a broad range of characters.
- exceptions make failures detectable.
- associative results give meaningful field names.
- a shared file avoids repeated credentials.
- the generic user message avoids exposing server details.

Check

- [ ] The connection file is reused.
- [ ] The database name is correct.
- [ ] The page works while MySQL is running.
- [ ] A stopped database produces a safe message.
- [ ] Credentials are excluded from screenshots and public repositories.

Using Prepared Statements

Prepared statements keep user-supplied values separate from SQL instructions. Use them whenever a value comes from a form, URL, session or uploaded file.

Unsafe pattern

```
$sql = "SELECT * FROM activities WHERE location = '" . $_GET["location"] . "'";
```

Concatenating input into SQL can allow SQL injection and can break on punctuation.

Safe pattern

```
$location = trim($_GET["location"] ?? "");

$stmt = $pdo->prepare(
    "SELECT activity_id, activity_name, location, capacity
     FROM activities
     WHERE location = :location"
);
$stmt->execute(["location" => $location]);
$activities = $stmt->fetchAll();
```

The placeholder is part of the SQL. The value is supplied separately.

Validate before querying

```
if ($location === "" || mb_strlen($location) > 100) {
    exit("Choose a valid location.");
}
```

Prepared statements address SQL injection; validation checks whether the data is acceptable for the application. Use both.

Common operations

```
// One placeholder
$stmt = $pdo->prepare("SELECT * FROM activities WHERE activity_id = :id");
$stmt->execute(["id" => $activityId]);

// Several placeholders
$stmt = $pdo->prepare(
    "UPDATE activities SET capacity = :capacity WHERE activity_id = :id"
);
$stmt->execute(["capacity" => $capacity, "id" => $activityId]);
```

Check

- [ ] No user value is concatenated into SQL.
- [ ] Placeholder names are meaningful.
- [ ] Every placeholder receives a value.
- [ ] Input is validated before execution.
- [ ] Displayed output is escaped with `htmlspecialchars()`.

Inserting Validated Form Data

An insert should accept input, validate it, store it with a prepared statement and report a useful result.

Example form

```
<form method="post">
  <label for="activity_name">Activity name</label>
  <input id="activity_name" name="activity_name" maxlength="100" required>

  <label for="capacity">Capacity</label>
  <input id="capacity" name="capacity" type="number" min="1" max="100" required>

  <button type="submit">Add activity</button>
</form>
```

Browser validation helps users, but PHP must validate again.

Process the submission

```
$name = trim($_POST["activity_name"] ?? "");
$capacity = filter_input(INPUT_POST, "capacity", FILTER_VALIDATE_INT);
$errors = [];

if ($name === "" || mb_strlen($name) > 100) {
    $errors[] = "Enter an activity name of 100 characters or fewer.";
}
if ($capacity === false || $capacity < 1 || $capacity > 100) {
    $errors[] = "Capacity must be from 1 to 100.";
```

```
}

if (!$errors) {
    $stmt = $pdo->prepare(
        "INSERT INTO activities (activity_name, capacity)
         VALUES (:name, :capacity)"
    );
    $stmt->execute(["name" => $name, "capacity" => $capacity]);
}
```

Output safely

Escape values when redisplaying them:

```
<?= htmlspecialchars($name, ENT_QUOTES, "UTF-8") ?>
```

Check

- [ ] Required fields have labels.
- [ ] Server-side validation handles missing and invalid values.
- [ ] A prepared statement performs the insert.
- [ ] Success appears only after the database operation succeeds.
- [ ] Invalid input remains unstored.

Selecting and Displaying Records

A useful output begins with a query and ends with accessible, escaped HTML.

Select only required fields

```
$stmt = $pdo->query(
    "SELECT activity_id, activity_name, location, capacity
     FROM activities
     ORDER BY activity_name"
);
$activities = $stmt->fetchAll();
```

Avoid `SELECT *` when the page needs only particular fields.

Display the result

```
<?php if (!$activities): ?>
  <p>No activities are available.</p>
<?php else: ?>
  <table>
    <thead>
      <tr>
        <th scope="col">Activity</th>
        <th scope="col">Location</th>
        <th scope="col">Capacity</th>
      </tr>
    </thead>
    <tbody>
      <?php foreach ($activities as $activity): ?>
        <tr>
```

```
          <td><?= htmlspecialchars($activity["activity_name"]) ?></td>
          <td><?= htmlspecialchars($activity["location"]) ?></td>
          <td><?= (int) $activity["capacity"] ?></td>
        </tr>
      <?php endforeach; ?>
    </tbody>
  </table>
<?php endif; ?>
```

The condition handles an empty dataset. The loop creates one row per record. Table headers support interpretation and accessibility.

Check

- [ ] The query returns only needed fields.
- [ ] Ordering supports the user’s task.
- [ ] Empty results have a clear message.
- [ ] Text is escaped and numbers are cast.
- [ ] Headers describe each column.

Filtering and Sorting Records

Filters help users reduce a dataset to relevant records. Sorts place those records in a useful order.

Filter with a prepared value

```
$location = trim($_GET["location"] ?? "");

$stmt = $pdo->prepare(
    "SELECT activity_name, location, capacity
     FROM activities
     WHERE location = :location
     ORDER BY capacity DESC, activity_name ASC"
);
$stmt->execute(["location" => $location]);
$rows = $stmt->fetchAll();
```

The first sort shows the largest capacity first. The second gives predictable alphabetical order when capacities match.

Allow a safe sort choice

SQL identifiers cannot be treated like ordinary bound values. Map a small set of allowed choices:

```
$sortChoice = $_GET["sort"] ?? "name";

$allowedSorts = [
    "name" => "activity_name ASC",
    "capacity_high" => "capacity DESC",
    "capacity_low" => "capacity ASC"
];

$orderBy = $allowedSorts[$sortChoice] ?? $allowedSorts["name"];
$sql = "SELECT activity_name, location, capacity
```

```
        FROM activities
        ORDER BY $orderBy";

$rows = $pdo->query($sql)->fetchAll();
```

Never place an unchecked URL value directly after `ORDER BY`.

Check

- [ ] Filters reflect a real user need.
- [ ] Filter values use placeholders.
- [ ] Sort options come from an allow-list.
- [ ] The default order is predictable.
- [ ] The output states what filter is active.

Updating Existing Records

An update changes an existing row. The application must identify the row, validate changes and prevent unintended updates.

Load the selected record

```
$id = filter_input(INPUT_GET, "id", FILTER_VALIDATE_INT);
if (!$id) {
    exit("Invalid activity.");
}

$stmt = $pdo->prepare(
    "SELECT activity_id, activity_name, capacity
     FROM activities
     WHERE activity_id = :id"
);
$stmt->execute(["id" => $id]);
$activity = $stmt->fetch();

if (!$activity) {
    exit("Activity not found.");
}
```

Apply a valid change

```
$capacity = filter_input(INPUT_POST, "capacity", FILTER_VALIDATE_INT);

if ($capacity === false || $capacity < 1 || $capacity > 100) {
    $error = "Capacity must be from 1 to 100.";
} else {
    $stmt = $pdo->prepare(
        "UPDATE activities
```

```
        SET capacity = :capacity
        WHERE activity_id = :id"
    );
    $stmt->execute(["capacity" => $capacity, "id" => $id]);
}
```

Always include a precise `WHERE` condition. An update without it can change every row.

Authorisation

A valid login does not automatically grant editing permission. Check the session role before showing or processing administrative actions.

Check

- [ ] Identifier is validated.
- [ ] Missing records are handled.
- [ ] New values are validated.
- [ ] Prepared statements are used.
- [ ] The update has a precise `WHERE` clause.
- [ ] Authorisation is checked on the server.