PHP and MySQL

Standalone tutorials for building secure PHP and MySQL web applications with PDO, validation, sessions and role-based access.

Login Systems

Build secure registration, login, sessions, protected pages and user/admin role-based access with PDO.

Creating a User Database and Table Using SQL

A login system needs a users table that can identify accounts, store password hashes and enforce roles. This example uses fictional accounts and prepares the database for PDO-based PHP pages.

> Never store plain-text passwords. The `password` field below stores the output from PHP's `password_hash()` function.

Create the database

Open `http://localhost/phpmyadmin/`, select **SQL**, and run:

```
CREATE DATABASE project_db
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

USE project_db;
```

Create the users table

```
CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL,
  role VARCHAR(20) NOT NULL DEFAULT 'user',
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT chk_user_role CHECK (role IN ('user', 'admin'))
);
```

Field	Purpose
`user_id`	Stable unique identifier
`username`	Unique login name
`password`	Password hash, never the original password
`role`	Authorisation level: `user` or `admin`
`created_at`	Account creation time

Some older MariaDB versions accept but do not enforce `CHECK` constraints. PHP must still validate the role, and ordinary registration must never accept an administrator role from the user.

Create a fictional administrator

Generate a hash with PHP's `password_hash()` or the classroom password-hasher tool. Insert the generated hash—not the test password:

```
INSERT INTO users (username, password, role)
VALUES ('admin_demo', '$2y$10$REPLACE_WITH_A_REAL_TEST_HASH', 'admin');
```

Use only a fictional test password. Never enter a personal, school or reused password into a classroom tool.

Verify the table

```
DESCRIBE users;
SELECT user_id, username, role, created_at FROM users;
```

Do not select or display password hashes unless diagnosing a specific local problem.

Check

- ☐ Username is unique.
- ☐ Password field is long enough for modern hashes.
- ☐ New accounts default to `user`.
- ☐ Administrator status cannot be selected during public registration.

☐ All accounts and data are fictional.

Secure Password Storage with Password Hashing

Password hashing converts a password into a one-way value suitable for storage. PHP automatically includes a salt, so the same password can produce different valid hashes.

> Use fictional test passwords only. Never enter your school, email, banking or reused personal password.

Generate a hash

```
<?php
$testPassword = "fictional-test-password";
$hash = password_hash($testPassword, PASSWORD_DEFAULT);

echo htmlspecialchars($hash, ENT_QUOTES, "UTF-8");
```

Store the complete hash in a `VARCHAR(255)` database field. Do not shorten it.

Verify a submitted password

```
<?php
$submittedPassword = "fictional-test-password";
$storedHash = '$2y$10$REPLACE_WITH_A_COMPLETE_HASH';

if (password_verify($submittedPassword, $storedHash)) {
    echo "Password accepted.";
} else {
    echo "Password not accepted.";
```

```
}
```

Login code should retrieve the stored hash by username and pass it to `password_verify()`. Never hash the submitted password again and compare strings; salts make that unreliable.

Rehash when needed

```
if (password_needs_rehash($storedHash, PASSWORD_DEFAULT)) {
    $newHash = password_hash($submittedPassword, PASSWORD_DEFAULT);
    // Update the stored hash using a prepared statement.
}
```

This allows PHP’s current default algorithm to improve over time.

Safe practice

- Do not print passwords.
- Do not place passwords or connection details in screenshots.
- Do not email or log submitted passwords.
- Use HTTPS on hosted systems.
- Use prepared statements for inserts and updates.
- Keep login failure messages generic.

Check

- ☐ Database stores hashes, not original passwords.
- ☐ `password_hash()` is used during registration.
- ☐ `password_verify()` is used during login.
- ☐ Hash field is `VARCHAR(255)`.
- ☐ Test data contains no real credentials.

Creating a User Registration Form

This standalone registration page accepts a fictional username and password, validates both on the server, hashes the password and inserts a standard `user` account with PDO.

Database connection

The example expects `includes/db.php` to create a PDO object named `$pdo`.

Registration page

```
<?php
require __DIR__ . "/includes/db.php";

$username = "";
$errors = [];
$created = false;

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $username = trim($_POST["username"] ?? "");
    $password = $_POST["password"] ?? "";
    $confirm = $_POST["confirm_password"] ?? "";

    if (!preg_match('/^[A-Za-z0-9_]{3,50}$/', $username)) {
        $errors[] = "Username must be 3–50 letters, numbers or underscores.";
    }
    if (strlen($password) < 10) {
        $errors[] = "Password must contain at least 10 characters.";
    }
    if ($password !== $confirm) {
```

```
        $errors[] = "Passwords do not match.";
    }

    if (!$errors) {
        $check = $pdo->prepare(
            "SELECT user_id FROM users WHERE username = :username"
        );
        $check->execute(["username" => $username]);

        if ($check->fetch()) {
            $errors[] = "That username is unavailable.";
        } else {
            $insert = $pdo->prepare(
                "INSERT INTO users (username, password, role)
                 VALUES (:username, :password, 'user')"
            );
            $insert->execute([
                "username" => $username,
                "password" => password_hash($password, PASSWORD_DEFAULT)
            ]);
            $created = true;
            $username = "";
        }
    }
}
?>
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Create account</title>
</head>
<body>
<h1>Create account</h1>

<?php if ($created): ?>
  <p role="status">Account created. You can now log in.</p>
<?php endif; ?>
```

```
<?php if ($errors): ?>
  <div role="alert">
    <p>Please correct the following:</p>
    <ul>
      <?php foreach ($errors as $error): ?>
        <li><?= htmlspecialchars($error) ?></li>
      <?php endforeach; ?>
    </ul>
  </div>
<?php endif; ?>

<form method="post">
  <label for="username">Username</label>
  <input id="username" name="username" maxlength="50"
         autocomplete="username" required
         value="<?= htmlspecialchars($username) ?>">

  <label for="password">Password</label>
  <input id="password" name="password" type="password"
         autocomplete="new-password" required>

  <label for="confirm_password">Confirm password</label>
  <input id="confirm_password" name="confirm_password" type="password"
         autocomplete="new-password" required>

  <button type="submit">Create account</button>
</form>
</body>
</html>
```

The form never accepts a role. Every public registration receives the server-controlled `user` role.

Test

Test valid input, short passwords, mismatched passwords, invalid usernames, duplicates and missing fields. Confirm that unsuccessful attempts create no database row.

Check

- [ ] Labels are associated with inputs.
- [ ] PHP validates all inputs.
- [ ] Username duplicates are handled.
- [ ] Password is hashed before insertion.
- [ ] Prepared statements are used.
- [ ] Registration cannot create an administrator.

Creating a Login Form

This standalone login page validates credentials with PDO, starts a secure session and redirects authenticated users. It uses one generic failure message so it does not reveal whether a username exists.

Login page

```
<?php
session_start();
require __DIR__ . "/includes/db.php";

$error = "";

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $username = trim($_POST["username"] ?? "");
    $password = $_POST["password"] ?? "";

    $stmt = $pdo->prepare(
        "SELECT user_id, username, password, role
         FROM users
         WHERE username = :username"
    );
    $stmt->execute(["username" => $username]);
    $user = $stmt->fetch();

    if ($user && password_verify($password, $user["password"])) {
        session_regenerate_id(true);
        $_SESSION["user_id"] = (int) $user["user_id"];
        $_SESSION["username"] = $user["username"];
        $_SESSION["role"] = $user["role"];

        header("Location: dashboard.php");
        exit;
```

```
    }

    $error = "Username or password was not accepted.";
}
?>
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Log in</title>
</head>
<body>
<h1>Log in</h1>

<?php if ($error): ?>
  <p role="alert"><?= htmlspecialchars($error) ?></p>
<?php endif; ?>

<form method="post">
  <label for="username">Username</label>
  <input id="username" name="username" autocomplete="username" required>

  <label for="password">Password</label>
  <input id="password" name="password" type="password"
         autocomplete="current-password" required>

  <button type="submit">Log in</button>
</form>
</body>
</html>
```

Why the security steps matter

- prepared statements keep input separate from SQL
- `password_verify()` checks the stored hash
- session ID regeneration reduces session fixation risk
- role comes from the database, not the form

- a generic error reduces account discovery

Test

Test a valid standard user, a valid administrator, a wrong password, an unknown username and empty fields. Confirm that failed attempts do not create session identity values.

Check

- [ ] Session starts before output.
- [ ] Query uses a prepared statement.
- [ ] Password hash is verified correctly.
- [ ] Session ID changes after login.
- [ ] User ID, username and role are stored.
- [ ] Failure message is generic.

Using PHP Sessions to Keep Users Logged In

PHP sessions store a small amount of trusted server-side state between requests. A login system can use them to remember the authenticated user’s identifier, username and role.

Start the session

Call `session_start()` before HTML or other output:

```
<?php
session_start();
```

Set identity after successful login

Only set these values after a database user has been found and `password_verify()` succeeds:

```
session_regenerate_id(true);

$_SESSION["user_id"] = (int) $user["user_id"];
$_SESSION["username"] = $user["username"];
$_SESSION["role"] = $user["role"];
```

The role must come from the database. Never trust a role supplied by a login or registration form.

Read session values safely

```
<?php
session_start();
```

```
$username = $_SESSION["username"] ?? "";
$role = $_SESSION["role"] ?? "user";
?>
<p>
  Signed in as
  <?= htmlspecialchars($username, ENT_QUOTES, "UTF-8") ?>
</p>
```

Session values still need escaping when inserted into HTML.

What belongs in a session

Suitable:

- internal user identifier
- display username
- authorised role
- short status messages
- CSRF tokens

Avoid:

- plain-text passwords
- complete database records
- sensitive information not needed across requests
- values copied directly from unvalidated form fields

Important limits

A session records authentication state; it does not automatically protect a page. Every protected server-side route must check the required identity and role.

Check

- [ ] Session starts before output.
- [ ] ID is regenerated after successful login.

- [ ] Identity and role come from the database.
- [ ] Password is never stored in the session.
- [ ] Session text is escaped when displayed.
- [ ] Protected routes perform their own checks.

Protecting Pages and Preventing Unauthorised Access

A protected page checks authentication on the server before sending restricted content. Hiding a menu link is not protection because a user can enter the URL directly.

Require a logged-in user

Place this before any HTML:

```
<?php
session_start();

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
```

The `exit` prevents the rest of the page from running after the redirect.

Complete protected page

```
<?php
session_start();

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
```

```
    exit;
}

$username = $_SESSION["username"] ?? "user";
?>
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Dashboard</title>
</head>
<body>
  <h1>User dashboard</h1>
  <p>Welcome, <?= htmlspecialchars($username) ?>.</p>
</body>
</html>
```

Use a reusable guard

Create `includes/require-login.php`:

```
<?php
if (session_status() !== PHP_SESSION_ACTIVE) {
    session_start();
}

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
```

Then begin protected pages with:

```
require __DIR__ . "/includes/require-login.php";
```

Test direct access

Use a private browser window or log out, then enter the protected URL directly. Also test after closing the browser and after destroying the session.

Check

- [ ] Authentication is checked before output.
- [ ] Redirect is followed by `exit`.
- [ ] Every protected processing route uses the guard.
- [ ] Direct URL access is tested.
- [ ] Displayed session values are escaped.

Creating a Logout Page

Logout should remove server-side session data, expire the session cookie and return the user to a safe public page.

Logout script

Create `logout.php`:

```
<?php
session_start();

$_SESSION = [];

if (ini_get("session.use_cookies")) {
    $parameters = session_get_cookie_params();

    setcookie(
        session_name(),
        "",
        time() - 42000,
        $parameters["path"],
        $parameters["domain"],
        $parameters["secure"],
        $parameters["httponly"]
    );
}

session_destroy();

header("Location: login.php");
exit;
```

Clearing `$_SESSION` removes values in the current request. Expiring the cookie removes the browser’s session identifier. `session_destroy()` removes the stored session.

Logout control

For a basic local classroom project:

```
<a href="logout.php">Log out</a>
```

For a stronger design, use a POST form and a CSRF token so another website cannot trigger logout unexpectedly.

Test

1. Log in and open a protected page.
2. Log out.
3. use the Back button and refresh.
4. enter the protected URL directly.
5. confirm the application requires login again.

Check

- [ ] Session array is cleared.
- [ ] Session cookie is expired when cookies are used.
- [ ] Stored session is destroyed.
- [ ] Redirect is followed by `exit`.
- [ ] Protected content is unavailable after logout.

Adding Role-Based Access Control

Role-based access control allows authenticated users to perform only the actions permitted by their stored role. This example uses two roles: `user` and `admin`.

Database role

A suitable users table contains:

```
role VARCHAR(20) NOT NULL DEFAULT 'user'
```

Ordinary registration must always create `user` accounts. Promote a fictional test account through a controlled administrator process or directly in the local development database:

```
UPDATE users
SET role = 'admin'
WHERE username = 'admin_demo';
```

Do not allow a public form to submit its own role.

Store the trusted role at login

After verifying the password:

```
session_regenerate_id(true);
$_SESSION["user_id"] = (int) $user["user_id"];
$_SESSION["username"] = $user["username"];
$_SESSION["role"] = $user["role"];
```

Require an administrator

```
<?php
session_start();

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}

if (($_SESSION["role"] ?? "") !== "admin") {
    http_response_code(403);
    exit("You do not have permission to access this page.");
}
```

Authentication asks “Who is signed in?” Authorisation asks “May that user perform this action?” Both checks are required.

Reusable administrator guard

Create `includes/require-admin.php` containing the checks above, then require it from every administrative display and processing route.

Test matrix

Account state	User page	Admin page
Logged out	Redirect to login	Redirect to login
Standard user	Allowed	403 response
Administrator	Allowed	Allowed
Changed form/URL value	No role change	No additional access

Check

- [ ] Roles are `user` and `admin`.
- [ ] Role comes from the stored user record.
- [ ] Administrative processing is protected.
- [ ] Direct URLs are tested.
- [ ] Denied access uses an appropriate response.

Creating a Navigation Menu Based on User Roles

Role-aware navigation shows users the actions available to them. It improves usability, but server-side checks remain responsible for security.

Start with a protected page

```
<?php
require __DIR__ . "/includes/require-login.php";

$username = $_SESSION["username"] ?? "user";
$role = $_SESSION["role"] ?? "user";
?>
```

Display navigation

```
<nav aria-label="Main navigation">
  <ul>
    <li><a href="dashboard.php">Dashboard</a></li>
    <li><a href="results.php">Results</a></li>

    <?php if ($role === "admin"): ?>
      <li><a href="import.php">Import dataset</a></li>
      <li><a href="manage-users.php">Manage users</a></li>
    <?php endif; ?>

    <li><a href="logout.php">Log out</a></li>
  </ul>
```

```
</nav>

<p>
  Signed in as
  <?= htmlspecialchars($username, ENT_QUOTES, "UTF-8") ?>
</p>
```

Protect every destination

The condition only controls whether the link appears. `import.php` and every other administrator route must also require `includes/require-admin.php`.

Do not use JavaScript or CSS visibility as an access-control mechanism. Those technologies run in the user’s browser and can be changed.

Design guidance

- Use descriptive link text.
- Identify the current page with `aria-current="page"`.
- Keep navigation order consistent.
- Do not display links that will always deny the current role.
- Provide a visible logout action.
- Ensure keyboard focus is clear in CSS.

Test

Compare logged-out, standard-user and administrator views. Then type each protected URL directly to verify that hidden links are not the only control.

Check

- [ ] Standard users see standard actions.
- [ ] Administrators see administrative actions.
- [ ] Every destination enforces access independently.
- [ ] Navigation is labelled and keyboard accessible.

- [ ] Session text is escaped.

Protecting an Admin-Only Data Import Page

A role-based system must enforce permissions on the server. Hiding a link is helpful navigation, but it does not protect the page.

Store trusted session values at login

After verifying the password and retrieving the database user:

```
session_regenerate_id(true);
$_SESSION["user_id"] = (int) $user["user_id"];
$_SESSION["role"] = $user["role"];
```

The role must come from the database, not from a form field supplied by the user.

Protect the import page

Place this code before any HTML output:

```
<?php
session_start();

if (!isset($_SESSION["user_id"])) {
    header("Location: login.php");
    exit;
}
```

```
if (($_SESSION["role"] ?? "") !== "admin") {
    http_response_code(403);
    exit("You do not have permission to access this page.");
}
```

Protect processing as well as display

The role check must run on the script that processes the uploaded CSV. A user can send a request directly even when the navigation link is hidden.

You may keep the form and processing in one protected file or require the same protection file from both scripts.

Show navigation by role

```
<?php if (($_SESSION["role"] ?? "") === "admin"): ?>
  <a href="import.php">Import dataset</a>
<?php endif; ?>
```

This improves usability but is not the security control.

Test the access rules

Test	Expected result
Logged out user opens import URL	Redirected to login
Standard user opens import URL	403 response
Administrator opens import URL	Upload form appears
Standard user submits directly	Request rejected
Changed browser role field	No effect because role comes from session

Check

- [ ] Sessions start before output.
- [ ] Login and role are checked server-side.
- [ ] Processing route repeats the protection.
- [ ] Roles come from stored user records.
- [ ] Tests include direct URL access.
- [ ] Test accounts and data are fictional.

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
<?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.