12.2 Authorization in PHP

🎓 What Is Authorization?

In our last lesson, we focused on **authentication** — verifying *who* a user is. Now that we know who someone is, we need to decide *what they're allowed to do.*

That's the role of **authorization**.

Authorization is about **permissions** — defining which users can view certain pages, perform certain actions, or access specific data.

In this video, we'll look at how to implement role-based authorization in PHP, how to restrict page access, and how to keep your logic clean and secure.

🧠 1. Understanding the Difference

Let's start with a quick recap:

Concept	Purpose	Stored Data
Authentication	Confirms *who* the user is	Session variables
Authorization	Controls *what* the user can do	Role or permission data

These two concepts usually work together: Once a user logs in successfully, PHP stores their ID and **role** in the session. Every page request after that can check their role to decide whether they're allowed to proceed.

⚙️ 2. How Authorization Fits into the Flow

A typical flow looks like this:

1. The user logs in.
2. The system looks up their role from the database — for example, "intern," "staff," or "admin."
3. PHP stores that role in `$_SESSION`.
4. Each protected page checks `$_SESSION['role']` to determine if access should be granted.

This model is called **Role-Based Access Control**, or **RBAC**. It's simple, clear, and works well for most applications.

🧩 3. Adding Roles to the Database

Let's start by making sure our users table supports roles.

```
ALTER TABLE users
ADD COLUMN role ENUM('intern', 'staff', 'admin')
```

```
NOT NULL DEFAULT 'intern';
```

Now every user has a defined role — which will determine their level of access.

🗒 4. Including the Role During Login

When users log in, we'll retrieve their role from the database and store it in the session along with their username and ID.

[Switch to code editor: `12-1-login.php`]

```
<?php
session_start();
require_once 'database.php';

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

    $pdo = Database::getConnection();
    $stmt = $pdo->prepare('SELECT id, password_hash, role FROM users WHERE
username = :username');
    $stmt->execute([':username' => $username]);
    $user = $stmt->fetch();

    if ($user && password_verify($password, $user['password_hash'])) {
        $_SESSION['userId'] = (int)$user['id'];
        $_SESSION['username'] = $username;
        $_SESSION['role'] = $user['role'];
        header('Location: 12-1-dashboard.php');
        exit;
    } else {
        echo "<p>Invalid username or password.</p>";
    }
}
?>
```

This change is small but important — it ensures that every page in our application can access the user's role to make decisions about what to display or allow.

🧱 5. Restricting Access to Pages

`12-2-admin-dashboard.php`

Now let's look at how to restrict a page to administrators only.

```
<?php
session_start();

// Step 1: Check if the user is logged in
if (!isset($_SESSION['userId'])) {
    header('Location: 12-1-login.php');
    exit;
}

// Step 2: Check if the user has the correct role
if (($_SESSION['role'] ?? '') !== 'admin') {
    http_response_code(403);
    echo "<h2>Access Denied</h2>";
    echo "<p>You do not have permission to view this page.</p>";
    exit;
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Admin Dashboard</title>
</head>
<body>
  <h2>Welcome, <?php echo htmlspecialchars($_SESSION['username']); ?>!</h2>
  <p>You have administrator access.</p>
</body>
</html>
```

This page uses a two-step check:

1. It ensures the user is logged in.
2. It verifies that their role is `admin`.

If either check fails, the page stops execution and returns a **403 Forbidden** response. This is a standard way to indicate that the user is authenticated but not authorized to access the resource.

🧭 6. Hiding Restricted Links in the Interface

Even though access is enforced on the server, it's a good idea to hide restricted links in your navigation. That way, users don't see options they can't use.

```
<?php if (($_SESSION['role'] ?? '') === 'admin'): ?>
  <p><a href="12-2-admin-dashboard.php">Admin Dashboard</a></p>
<?php endif; ?>
```

This doesn't replace your server-side checks — it just improves the user experience by showing relevant navigation based on role.

🗝️ 7. Role-Based vs. Permission-Based Access

So far we've been using **Role-Based Access Control (RBAC)** — where each role has a predefined set of permissions. This works well for small to medium applications.

For larger systems, you might use **Permission-Based Access**, where each user or role is assigned individual privileges, such as:

```
$permissions = [
    'canApprove' => true,
    'canDelete' => false,
];
```

The logic stays the same — PHP checks these flags before allowing an action.

⚙️ 8. Modern PHP Enhancements

Recent versions of PHP give us new features that make this logic cleaner and safer:

- **Strict typing** helps ensure roles are always strings, not numbers or nulls.
- `match` **expressions** simplify multi-role routing:

```
$redirectTo = match ($_SESSION['role'] ?? '') {
    'admin' => 'admin-dashboard.php',
    'staff' => 'staff-dashboard.php',
    default => 'intern-dashboard.php',
};
header("Location: $redirectTo");
exit;
```

- **Named arguments and exceptions** in PDO make database queries easier to read and maintain.

These features help keep your authorization code both clear and reliable.

🧠 9. Demonstration: Testing Access

Let's test it:

1. Log in as an *intern* — you'll be redirected to your basic dashboard.
2. Try navigating to `12-2-admin-dashboard.php` manually — you'll get a 403 Access Denied.
3. Log out, then log in as an *admin* — this time, you'll have full access.
4. Note that only admins see the Admin Dashboard link in the navigation.

That's role-based authorization in action.

✅ 10. Wrap-Up

So to summarize:

- **Authentication** identifies *who* the user is.
- **Authorization** determines *what* they can do.
- We store each user's **role** in their session.
- Each protected page checks that role before granting access.
- PHP 8 features like `match` and strict typing help make this cleaner and safer.

In our next lesson, we'll continue building on this by securing our forms and actions against **CSRF attacks** — another crucial step in keeping your web applications safe.