# 12.1 Authentication in PHP

## 🎓 What Is Authentication?

In this video, we're going to talk about **authentication** — the process of verifying a user's identity.

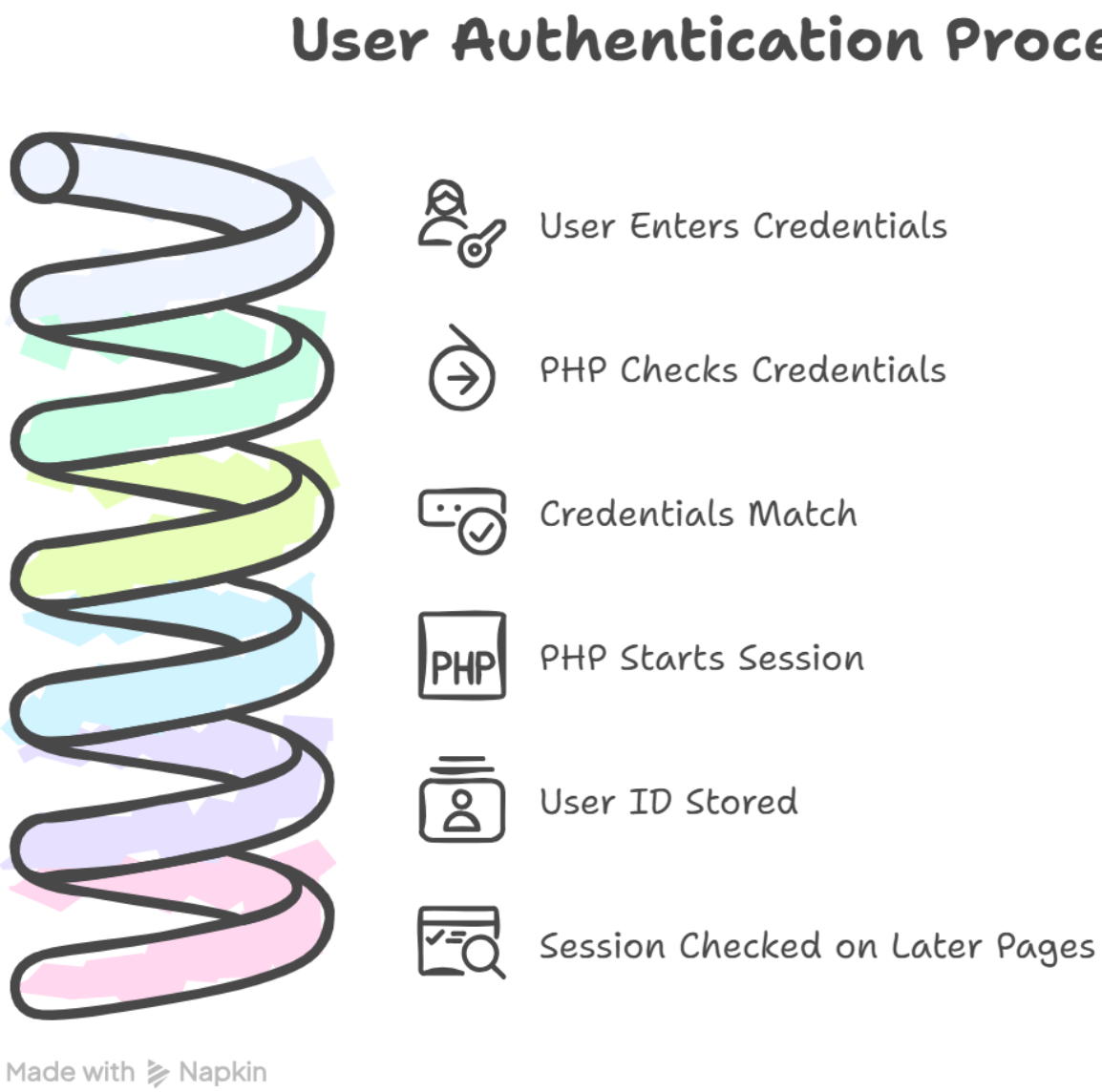
In other words, authentication answers one question: **"Who are you?"**

Once we know who a user is, **authorization** determines **what they're allowed to do** — but we'll cover that later.

Right now we'll focus on **how PHP handles authentication** in modern applications, including secure password hashing and session-based logins.

---

## 🧠 1. How Authentication Works in PHP



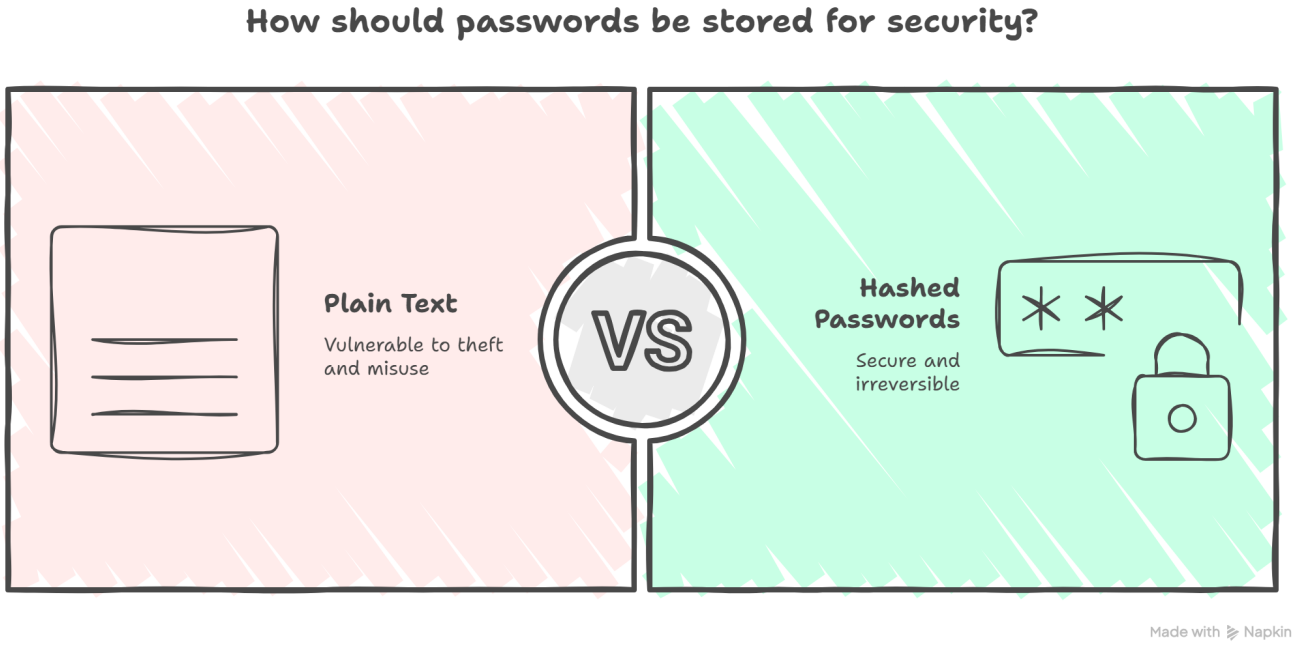

Most authentication systems follow this pattern:

1. The user enters a **username and password**.
2. PHP checks those credentials against stored data in a **database**.
3. If they match, PHP starts a **session** and stores the user's ID in `$_SESSION`.
4. On later pages, PHP checks the session to confirm the user is still authenticated.

This process connects user identity to **state management** — because once logged in, we need to remember that identity across requests.

---

## 🔐 2. Password Storage – Why Hashing Matters




In older systems, passwords were sometimes stored in plain text — that's a serious security risk. In modern PHP, we never store passwords directly. We store a **hash** — a one-way, irreversible representation.

PHP makes this easy using built-in functions:

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

`PASSWORD_DEFAULT` currently uses **bcrypt** by default, but PHP automatically upgrades this algorithm as newer versions become available.

When the user logs in, we verify like this:

```
if (password_verify($password, $hash)) {
    echo "Password is valid!";
}
```

This means even if the database is compromised, attackers can't easily reverse the hashes to recover passwords.

---

## ⚙️ 3. Example Project: Simple Login System

Let's build a simple login system that authenticates users using a database and securely hashed passwords.

We'll create a simple MySQL table named `users` like this:

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
```

```
  username VARCHAR(100) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL
);
```

---

### Step 1 – Registration (storing a new user)

`12-1-register.php`

```
<?php
declare(strict_types=1);
require_once '12-1-database.php';

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

    if ($username && $password) {
        $pdo = Database::getConnection();
        $hash = password_hash($password, PASSWORD_DEFAULT);

        $stmt = $pdo->prepare('INSERT INTO users (username, password_hash) VALUES
(:username, :hash)');
        $stmt->execute([':username' => $username, ':hash' => $hash]);

        echo "<p>Registration successful. <a href='12-1-login.php'>Log in</a>
</p>";
    } else {
        echo "<p>Please provide a username and password.</p>";
    }
}
?>
<form method="post">
  <p>Username: <input type="text" name="username" required></p>
  <p>Password: <input type="password" name="password" required></p>
  <p><input type="submit" value="Register"></p>
</form>
```

When a user registers, we hash the password using `password_hash()` before saving it. This ensures that even the site administrator can't see the raw passwords.

---

### Step 2 – Logging In

`12-1-login.php`

```
<?php
declare(strict_types=1);
session_start();
```

```php
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 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;

        // Check if the hash needs rehashing (optional)
        if (password_needs_rehash($user['password_hash'], PASSWORD_DEFAULT)) {
            $newHash = password_hash($password, PASSWORD_DEFAULT);
            $updateStmt = $pdo->prepare('UPDATE users SET password_hash = :hash
WHERE id = :id');
            $updateStmt->execute([':hash' => $newHash, ':id' => $user['id']]);
        }

        header('Location: 12-1-dashboard.php');
        exit;
    } else {
        echo "<p>Invalid username or password.</p>";
    }
}
?>
<form method="post">
  <p>Username: <input type="text" name="username" required></p>
  <p>Password: <input type="password" name="password" required></p>
  <p><input type="submit" value="Login"></p>
</form>
```

Here we:

1. Start a session.
2. Retrieve the stored hash for the username.
3. Verify the password with `password_verify()`.
4. Save the user's ID and username in `$_SESSION` if the login succeeds.

This session variable now identifies the authenticated user for future requests.

---

### Step 3 – Protecting Pages

**12-1-dashboard.php**

```php
<?php
declare(strict_types=1);
session_start();

if (!isset($_SESSION['userId'])) {
    header('Location: 12-1-login.php');
    exit;
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Dashboard</title>
</head>
<body>
<h2>Welcome, <?php echo htmlspecialchars($_SESSION['username']); ?>!</h2>
<p>This is your dashboard. Only logged-in users can see this.</p>
<p><a href="12-1-logout.php">Logout</a></p>
</body>
</html>
```

The first thing `12-1-dashboard.php` does is check if `$_SESSION['userId']` is set. If not, it redirects to the login page. This is the standard pattern for protecting pages in PHP applications.

---

### Step 4 – Logging Out

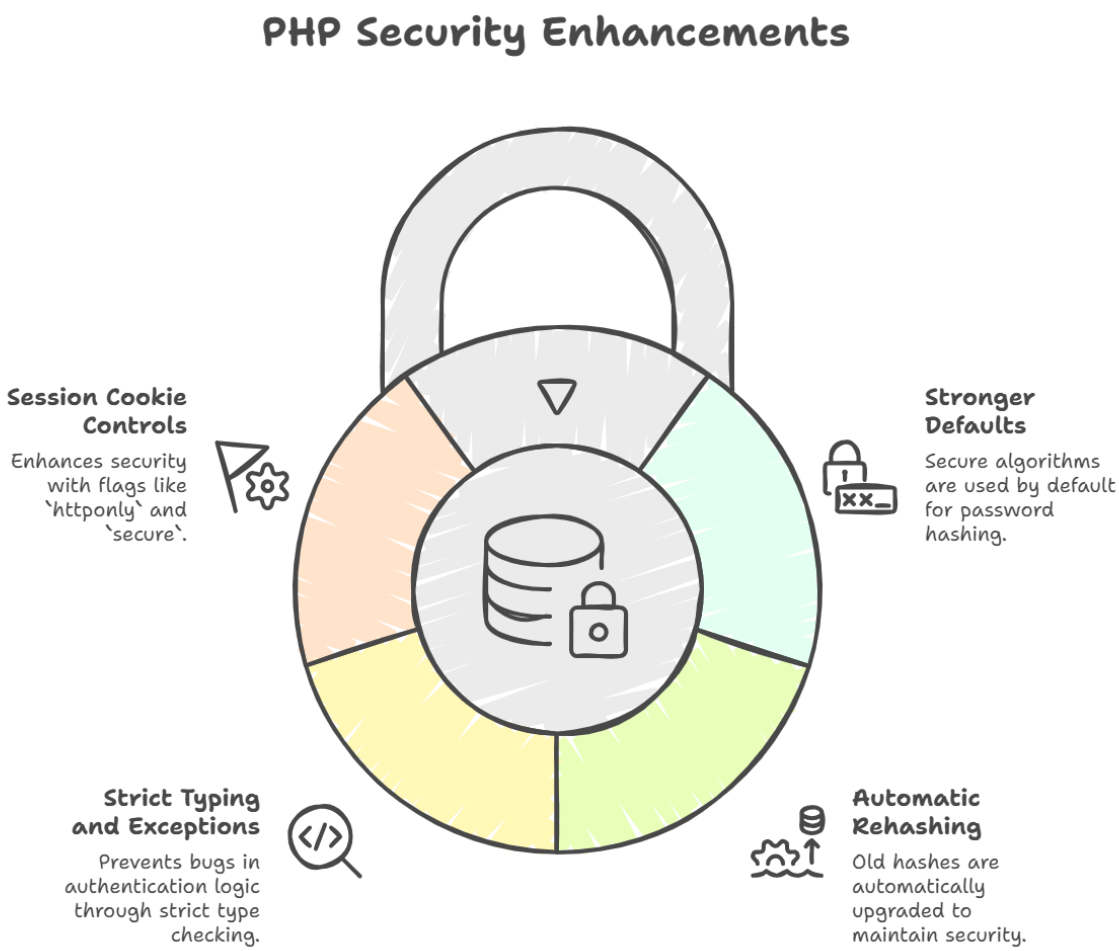
**`12-1-logout.php`**

```php
<?php
session_start();
session_unset();
session_destroy();
header('Location: 12-1-login.php');
exit;
```

Logging out simply means ending the session and redirecting the user.

---

## 🔓 4. Recent PHP Features That Strengthen Authentication




Recent versions of PHP bring several improvements that make authentication safer and easier to manage:

- **Stronger defaults**: `password_hash()` uses secure algorithms by default.
- **Automatic rehashing**: You can check with `password_needs_rehash()` to upgrade old hashes if PHP's defaults change.
- **Strict typing** and **exceptions** help prevent subtle bugs in authentication logic.
- **Session cookie controls** let you tighten security using `session_set_cookie_params()` for `httponly`, `secure`, and `samesite` flags.

---

## 🧭 5. Seeing Authentication in Action

Let's test it out:

1. Register a new user.
2. Log in — notice that a `PHPSESSID` cookie is created.
3. Open the **Application** → **Cookies** tab in Chrome DevTools to view it.

4. Navigate to `dashboard.php`.
    - You'll stay logged in because the session persists.
5. Click "Logout" — the session is destroyed and the cookie becomes invalid.

This round trip between client, session ID, and server is the core of PHP-based authentication.

---

## ☑ 6. Wrap-Up

So, to recap:

- Authentication identifies who the user is.
- We use secure **password hashing** to protect credentials.
- We use **sessions** to maintain authentication state.
- PHP 8.3 gives us strong, built-in tools to do this safely.

In our next video, we'll explore **authorization** — how to control access once the user is logged in.

Later, we'll extend this even further with **CSRF protection** and perhaps **token-based authentication** using **JSON Web Tokens (JWTs)**.