12.4 Using JSON Web Tokens (JWTs) for Authentication in PHP

🎓 The Problem with Sessions

So far, we've built secure, session-based authentication — and it works great for traditional PHP sites.

But in modern web development — especially when you're working with **single-page apps** or **mobile clients** — sessions can become difficult to scale.

That's where **JSON Web Tokens**, or **JWTs**, come in.

In this video, we'll explore what JWTs are, how they differ from sessions, and how to use them in PHP to create secure, stateless authentication.

🧠 1. What Are JSON Web Tokens (JWTs)?

A **JSON Web Token** is a self-contained, signed piece of data that proves who a user is.

Instead of storing session data on the server, the server sends the client a token that includes:

- The user's ID
- Their role or permissions
- An expiration time
- A digital signature

Each time the client makes a request, it sends that token back. The server verifies the signature to confirm it hasn't been tampered with — no session storage required.

🧩 2. JWT Structure

A JWT has three parts, separated by dots:

```
header.payload.signature
```

Example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1c2VySWQiOjEsInVzZXJuYW1lIjoiYWxleCJ9.
kNqpxx-r5v8hO7K5aR6D6aRZ0hz5TJKk0INxX5sWQoI
```

- **Header:** algorithm and token type
- **Payload:** the user's claims (like ID and role)

- **Signature:** verifies that the data hasn't been altered

JWTs are **signed**, not encrypted — meaning anyone can read the payload, but only the server can verify or create valid tokens.

⚙️ 3. How JWT Authentication Works

Here's the flow:

1. The user logs in with a username and password.
2. PHP verifies their credentials and generates a JWT containing their ID and role.
3. The server sends that token back to the client.
4. The client stores it — usually in **localStorage** or **sessionStorage**.
5. On every request, the client includes the token in the HTTP `Authorization` header.
6. The server verifies the token's signature and expiration time before granting access.

This creates a fully **stateless** authentication system — ideal for APIs and modern front-end frameworks.

🧰 4. Installing a JWT Library

While PHP doesn't include JWT handling natively, we can use the excellent open-source package `firebase/php-jwt`.

Install it with Composer:

```
composer require firebase/php-jwt
```

This library makes it easy to encode, sign, and verify JWTs.

📝 5. Creating a Token After Login

`12-4-config.php`

```
<?php
define('JWT_SECRET', 'your-secret-key-here');
```

This file holds our secret key used to sign the tokens.

`12-4-login.php`

Let's build a login endpoint that returns a JWT instead of starting a session.

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

```php
require_once '12-4-config.php';
require_once '12-1-database.php';
require_once '../vendor/autoload.php';

use Firebase\JWT\JWT;
use Firebase\JWT\Key;

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'])) {
        $payload = [
            'userId' => (int)$user['id'],
            'username' => $username,
            'role' => $user['role'],
            'exp' => time() + 3600 // expires in 1 hour
        ];

        $jwt = JWT::encode($payload, JWT_SECRET, 'HS256');

        header('Content-Type: application/json');
        echo json_encode(['token' => $jwt]);
        exit;
    }

    http_response_code(401);
    echo json_encode(['error' => 'Invalid credentials']);
}
```

Instead of starting a session, we create a JWT payload and sign it using our secret key. Then we send it to the client as JSON.

This token now represents the user's authenticated state.

🧩 6. Validating the Token

12-4-dashboard.php

Here's how we verify the token for a protected API endpoint.

```php
<?php
declare(strict_types=1);
require_once '12-4-config.php';
```

```php
require_once '../vendor/autoload.php';

use Firebase\JWT\JWT;
use Firebase\JWT\Key;

$headers = getallheaders();
$authHeader = $headers['Authorization'] ?? '';

if (!str_starts_with($authHeader, 'Bearer ')) {
    http_response_code(401);
    exit('Missing or invalid token');
}

$jwt = substr($authHeader, 7);

try {
    $decoded = JWT::decode($jwt, new Key(JWT_SECRET, 'HS256'));
    echo "Welcome, " . htmlspecialchars($decoded->username);
} catch (Exception $e) {
    http_response_code(401);
    exit('Invalid or expired token');
}
```

This script checks for the `Authorization` header, extracts the token, and verifies it using the same secret key. If it's valid, we can safely use the claims inside the token — like username or role — without needing any session data.

💾 7. Storing and Using Tokens in the Browser

[Switch to front-end JavaScript snippet]

On the client side, we'll store the JWT after a successful login and include it with future requests.

`12-4-login.html`

```html
<html>
<head>
  <title>Login</title>
</head>
<body>
<form id="loginForm">
  <input type="text" name="username" placeholder="Username" required>
  <input type="password" name="password" placeholder="Password" required>
  <button type="submit">Login</button>
</form>
<script>
document.getElementById('loginForm').addEventListener('submit', function(event) {
  event.preventDefault();
  fetch('12-4-login.php', {
    method: 'POST',
```

```
    body: new FormData(document.querySelector('form'))
  })
    .then(res => res.json())
    .then(data => {
      sessionStorage.setItem('jwt', data.token);
      window.location.href = '12-4-dashboard.html';
    });
});
</script>
</body>
</html>
```

For requests that require authentication:

`12-4-dashboard.html`

```
<html>
<head>
  <title>Dashboard</title>
</head>
<body>
<script>
const token = sessionStorage.getItem('jwt');

fetch('12-4-dashboard.php', {
  headers: { 'Authorization': 'Bearer ' + token }
})
  .then(res => res.text())
  .then(html => document.writeln(html));
</script>
</body>
</html>
```

Here we're using `sessionStorage`, which automatically clears when the browser tab is closed. If you want persistence between sessions, you could use `localStorage` instead — just be careful not to store tokens long-term unless absolutely necessary.

🔐 8. Security Best Practices

- Always use **HTTPS** — JWTs are just as sensitive as passwords.
- Set short token lifetimes (1 hour is typical).
- Use a **refresh token** system for long-lived sessions.
- Never store tokens in cookies to avoid CSRF exposure.
- Keep your signing secret safe and never hard-code it in client-side code.

These rules help keep token-based systems just as secure as traditional session systems — and often more scalable.

⚙️ 9. Comparing JWTs and PHP Sessions

Feature	JWT	PHP Session
Storage	Client-side	Server-side
State	Stateless	Stateful
Scalability	Excellent	Requires shared storage
Validation	Signature	Session lookup
Best Use	APIs, SPAs, mobile	Classic PHP apps

JWTs shine when you're building APIs or connecting to front-end frameworks like React, Vue, or Angular. But for traditional multi-page PHP applications, sessions are often simpler and safer.

Understanding both approaches lets you choose the right one for each project.

🧠 10. Demo Recap

[Switch to browser demonstration]

Let's test this workflow:

1. Submit the login form — the server responds with a JWT.
2. Open DevTools → Application → Storage → Session Storage to see it saved.
3. Refresh or navigate — the app includes the token automatically in the request header.
4. After an hour, the token expires, and access is denied until you log in again.

That's a complete, stateless authentication loop.

☑️ 11. Wrap-Up

So to summarize:

- **JWTs** allow secure, stateless authentication — no sessions needed.
- They're ideal for APIs, SPAs, and mobile apps.
- PHP 8.3 works beautifully with modern JWT libraries like `firebase/php-jwt`.
- Tokens must be stored, transmitted, and validated carefully.

By now, you've seen how to manage state in multiple ways — from cookies and sessions to CSRF protection and now JWTs. Together, these tools form a complete picture of secure, modern web development in PHP.