**Cookie-Authentifizierung in ASP.NET Core MVC**

|  |  |
| --- | --- |
| Begriff | Erklärung |
| Authentifizierung | Prüfung, ob jemand ist, wer er vorgibt zu sein (z. B. Login mit Passwort) |
| Autorisierung | Entscheidung, ob jemand auf eine **Ressource** zugreifen darf |
| Cookie | Kleine Datei im Browser, speichert Sitzungs- oder Anmeldeinformationen |
| Claim | Aussage über einen Benutzer (z. B. Name, **Rolle**, Abteilung) |
| Principal | Die Benutzeridentität (mit Claims), die im System verwendet wird |

**Cookie-Authentifizierung in ASP.NET Core**

1. Benutzer ruft Login-Seite auf

2. Benutzer gibt gültige Anmeldedaten ein

3. Server erstellt **ClaimsIdentity** + Cookie

4. Cookie wird im Browser gespeichert

5. Bei jedem Request sendet der Browser das Cookie mit

6. Middleware liest das Cookie aus → **ClaimsPrincipal** wird erstellt

7. **[Authorize]** prüft, ob der Benutzer authentifiziert (und ggf. autorisiert) ist

**Warum Cookies?**

* Cookies sind **zustandsbehaftet**, aber HTTP ist **zustandslos**.
* Cookies ermöglichen, **Benutzerdaten zwischen Anfragen** zu speichern.
* Alternativen: Session-IDs, JWTs ( = JSON Web Token)

**Funktionsweise in ASP.NET Core**

**Middleware-Konzept**

ASP.NET Core funktioniert **middleware-basiert**. Für Authentifizierung gibt es zwei entscheidende Komponenten:

1. **UseAuthentication()** – prüft, ob ein gültiger Benutzer vorhanden ist (Cookie vorhanden und gültig?)
2. **UseAuthorization()** – prüft, ob der Benutzer Zugriff auf die Ressource hat (z. B. per [Authorize])

**Claims-basierte Identität**

* In ASP.NET Core ist der Benutzer kein einfacher „Username“, sondern eine **Identität mit Claims**:
  + z. B. Name, Rolle, E-Mail, Abteilung
* Der **ClaimsPrincipal** wird im HttpContext.User gespeichert.

**Beispielhafte Claims (rein theoretisch):**

Name: admin

Role: Admin

Department: IT

Diese Claims entstehen beim Login – durch Prüfung der Anmeldedaten und Aufbau einer ClaimsIdentity.

**Autorisierung mit Rollen und Policies**

**[Authorize]-Attribut**

* [Authorize]: nur authentifizierte Benutzer dürfen auf die Methode zugreifen.
* [Authorize(Roles = "Admin")]: nur Benutzer mit Rolle „Admin“.
* [AllowAnonymous]: explizit Zugriff ohne Authentifizierung.

**Role vs. Policy**

| **Technik** | **Beispiel** | **Anwendungsfall** |
| --- | --- | --- |
| Rolle | [Authorize(Roles = "Admin")] | Einfache Rechte auf Basis von Rollen |
| Policy | [Authorize(Policy = "NurIT")] | Komplexere Prüfungen z. B. auf Claims |

**Beispielhafte Policy:**

Policy "NurIT": Benutzer muss Claim "Department" mit Wert "IT" besitzen.

**Ablauf einer Authentifizierung in ASP.NET Core**

1. Benutzer sendet Login-Daten an Controller

2. Backend prüft Zugangsdaten

3. Backend erstellt ClaimsIdentity → ClaimsPrincipal

4. ClaimsPrincipal wird in ein Cookie serialisiert

5. Cookie wird dem Client als HTTP-Response-Header gesendet

6. Bei Folgeanfragen sendet der Client das Cookie automatisch mit

7. Middleware prüft Cookie und setzt User-Information

8. Autorisierungsattribute steuern Zugriff auf Aktionen

**Standardverhalten von Auth-Cookie**

Wenn nichts explizit konfiguriert ist:

Das Cookie wird bei Session-Ende (Browser schließen) gelöscht – es ist ein nicht-persistentes Cookie.

Der Benutzer bleibt also nur während der aktuellen Sitzung angemeldet.

Konfiguration eines festen Ablaufdatums (persistentes Cookie)

In der Program.cs oder Startup.cs kann man die Lebensdauer so festlegen:

builder.Services.AddAuthentication("MyCookieScheme")

.AddCookie("MyCookieScheme", options =>

{

options.LoginPath = "/Account/Login";

options.ExpireTimeSpan = TimeSpan.FromMinutes(30); // Lebensdauer: 30 Minuten

options.SlidingExpiration = true; // Verlängert das Cookie bei Aktivität

});

**User in serverseitiger Session speichern?**

Die Verwendung von **Session** für Login-Funktionalitäten in Webanwendungen **ist nicht grundsätzlich "schlecht"**, aber **Warum Session-basierter Login problematisch sein kann?**

**1. Skalierung und Load-Balancing**

* In modernen Webanwendungen mit **mehreren Servern (Web-Farm)** kann die Session nicht ohne Weiteres zwischen Servern geteilt werden.
* Lösung: Sticky Sessions, Session-State-Server oder verteilte Caches (z. B. Redis) – das macht alles komplexer.
* → **Problematisch in Cloud-Umgebungen oder Microservices.**

**2. Abhängigkeit vom Serverzustand (Stateful Design)**

**Sessions sind serverseitig zustandsbehaftet**, was gegen das Prinzip von **REST (stateless)** verstößt.

Moderne APIs (z. B. mit ASP.NET Core Web API) bevorzugen **Token-basierte Authentifizierung (z. B. JWT)**.

**3. Session Timeout**

Wenn die Session abläuft, ist der Benutzer plötzlich ausgeloggt – oft ohne Vorwarnung.

Bei langer Inaktivität kann das zu **verlorener Arbeit** oder **Verwirrung** führen.

**4. Sicherheitsrisiken**

**Session Fixation**: Angreifer erzwingen Session-IDs im Vorfeld.

**Session Hijacking**: Wenn Session-IDs nicht sicher übertragen werden (z. B. ohne HTTPS), können sie abgefangen werden.

**CSRF-Anfälligkeit** ist bei Sessions höher, da keine Signatur wie bei JWT enthalten ist.

**5. Wartbarkeit und Debugging**

* Session-Daten sind **nicht sichtbar** im Netzwerkverkehr (nur die ID) → schwerer zu debuggen als z. B. ein JWT.