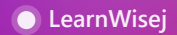


# Integration von OAuth und OpenID Connect

Lektionsleitfaden · Lab- und Prüfungsleitfaden



## Lektionsleitfaden

Den Login an einen Identity Provider delegieren, der Authorization Code Flow, der Umgang mit Redirects in einer Wisej.NET-App, Token-Validierung und die Abbildung von Claims auf den SessionContext. Was das für Entwickler bedeutet, die eine Wisej.NET-App im Produktivbetrieb absichern – und wie Sie dieses Modul lesen.



### Fokus: Produktivbetrieb

Den Login an einen Identity Provider delegieren, der Authorization Code Flow, der Umgang mit Redirects in einer Wisej.NET-App, Token-Validierung und die Abbildung von Claims auf den SessionContext.

## 1. Das Problem im Produktivbetrieb

AccessOps kann bereits ein eigenes Passwort prüfen, doch die Organisation, die es betreibt, will kein zweites Verzeichnis: Die Belegschaft meldet sich überall sonst beim Identity Provider des Unternehmens an, und Eintritte, Austritte und Multi-Faktor-Richtlinien werden an einer Stelle verwaltet. Den Login zu delegieren entfernt das Passwort vollständig aus AccessOps und ersetzt es durch eine andere Pflicht: Das Portal bekommt nun von einem Dritten über den Browser des Benutzers eine Identität übergeben und muss belegen, dass diese Identität echt, unverändert und für das Portal bestimmt ist, bevor auch nur ein einziger Abteilungsdatensatz angezeigt wird.

## 2. Zentrale Konzepte

| Konzept                           | Warum es wichtig ist                                                                                                                                                                                                                                              |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Authorization Code Flow           | Der OpenID-Connect-Ablauf, bei dem der Browser immer nur einen kurzlebigen Einmal-Code transportiert, während Ihr Server die Tokens über einen direkten HTTPS-Aufruf abholt, den der Browser nie sieht.                                                           |
| PKCE                              | Ein zufälliger Code Verifier, der pro Anmeldung erzeugt, an den Authorize-Endpoint nur als SHA-256-Hash gesendet und erst am Token-Endpoint offengelegt wird, sodass ein abgefangener Code von niemand anderem eingelöst werden kann.                             |
| ID-Token                          | Ein signiertes JWT, das bestätigt, dass dieser Provider dieses Subjekt für diesen Client zu diesem Zeitpunkt authentifiziert hat. Es ist das einzige Token, das AccessOps heranzieht, um zu entscheiden, wer der Benutzer ist.                                    |
| Access- und Refresh-Token         | Das Access-Token ist ein Bearer-Credential für den Aufruf der API eines anderen; das Refresh-Token beschafft im Back-Channel Ersatz. Keines von beiden ist ein Identitätsnachweis für Ihre eigene App.                                                            |
| State und Nonce                   | Zwei Zufallswerte, die Sie pro Anmeldung erzeugen: State bindet den Callback an die Anfrage, die Sie tatsächlich gestartet haben, Nonce bindet das zurückgelieferte ID-Token an genau diese Anfrage, sodass kein älteres Token darin wiederverwendet werden kann. |
| Redirect-URI                      | Die exakte, vorab registrierte URL, zu der der Provider den Browser zurückschickt. In AccessOps ist das ein kleiner serverseitiger Endpunkt statt eines Wisej.NET-Fensters, weil er laufen muss, bevor irgendeine UI existiert.                                   |
| Claim-Mapping und Provisionierung | Provider-Claims werden zu einem AccessOps-Mitarbeiter: Das Paar aus Issuer und Subject identifiziert das Konto, und bei der ersten Anmeldung wird eine lokale Zeile angelegt, die keine Berechtigungen trägt, bis jemand sie vergibt.                             |

## 3. Der rote Faden der Lektion

### Drei Tokens und ein Hin und Zurück

Alles beginnt beim Discovery-Dokument des Providers, der Well-known-Konfigurations-URL, die die Authorize-, Token- und End-Session-Endpunkte sowie die JWKS-Adresse seiner Signaturschlüssel auflistet. AccessOps liest es einmal beim Start und speichert es zwischen, denn jeder spätere Schritt hängt von den exakten veröffentlichten Werten ab – allen voran vom Issuer-String, der Zeichen für Zeichen übereinstimmen muss.

Die Anmeldung selbst besteht aus zwei halben Gesprächen. Im Front-Channel wird der Browser zum Authorize-Endpoint geschickt, mit Client-ID, registrierter Redirect-URI, `response_type=code`, den Scopes, die AccessOps braucht ( `openid profile email` ), einem frischen State, einer frischen Nonce und einer Code Challenge mit `code_challenge_method=S256`. Der Benutzer authentifiziert sich beim Provider – AccessOps sieht diese Seite nie und sieht auch das Passwort nie –, und der Browser kehrt zur Redirect-URI zurück und bringt nur einen Code und den State mit. Im Back-Channel sendet Ihr Server diesen Code

zusammen mit dem Code Verifier und dem Client-Credential an den Token-Endpunkt und erhält die Tokens im Response-Body.

Jedes Token hat genau eine Aufgabe. Das ID-Token ist die Authentifizierungsquittung und das Einzige, was beantwortet, wer sich angemeldet hat. Das Access-Token ist ein Credential für eine Ressourcen-API wie den Verzeichnis-Graph und darf in Ihrer eigenen App nie ausgewertet werden, um über die Identität zu entscheiden; es kann opak sein und ist an eine andere Audience adressiert. Das Refresh-Token beschafft neue Tokens ohne einen weiteren Umweg über den Browser. PKCE macht diese Aufteilung sicher: Ohne den Verifier ist ein Code, der aus einem Browserverlauf, einem Referrer-Header oder einem kompromittierten Proxy gestohlen wurde, für seinen Besitzer wertlos.

### Die Wisej.NET-Session verlassen und den Weg zurück finden

In AccessOps startet der Redirect im Servercode mit `Application.Navigate`, das den Browser auf die zusammengesetzte Authorize-URL schickt. Der Browser hat Ihre Anwendung nun verlassen und kehrt zur Redirect-URI zurück – und die kann kein Wisej.NET-Fenster sein, denn sie muss einen HTTP-Request beantworten, der eintrifft, bevor irgendein Fenster existiert. Der Callback ist hier ein kleiner serverseitiger Endpunkt in derselben Anwendung: ein eigener Wisej.NET-HTTP-Handler oder ein begleitender ASP.NET-Core-Endpunkt auf demselben Origin. Seine gesamte Aufgabe sind der Token-Austausch, die Validierung und ein einziger Redirect.

Der Faden zurück zum Kontext des Benutzers ist der State-Wert. Vor der Navigation erzeugt AccessOps ein zufälliges Handle und legt auf dem Server einen Datensatz für die ausstehende Anmeldung dazu ab: den Code Verifier, die Nonce, den Erstellungszeitpunkt und das Ziel, das der Benutzer ansteuern wollte. Nur das Handle reist im Query-String. Der Callback schlägt es nach, weist alles Unbekannte, Abgelaufene oder bereits Verwendete ab und löscht den Datensatz, damit er nicht zweimal eingelöst werden kann. Nichts anderem in diesem Query-String wird vertraut, denn der gesamte Front-Channel steht unter der Kontrolle des Browsers.

Sobald die Tokens validiert sind, leitet der Handler mit einem Einmal-Ticket zur Start-URL von AccessOps zurück; die Anwendung liest dieses Ticket beim Start aus dem Query-String des Requests, löst es genau einmal gegen denselben Speicher ein und baut den SessionContext aus dem Datensatz auf, nicht aus irgendetwas, das der Browser mitgebracht hat. Diese Indirektion hält die Session ehrlich, egal ob die ursprüngliche Session noch lebt oder der Rückweg eine neue gestartet hat. Den Provider in einem zweiten Browserfenster zu öffnen und der wartenden Session mit `Application.Update` ein Signal zu geben, ist eine ebenso gültige Variante; die Regel ändert sich nicht.

### Das ID-Token validieren und dann abbilden

Ein JWT ist signiert, nicht geheim: Jeder kann es dekodieren, daher ist das Lesen von Claims vor der Validierung der klassische Fehler. AccessOps prüft fünf Dinge mit `System.IdentityModel.Tokens.Jwt` und den `Microsoft.IdentityModel`-Paketen: die Signatur gegen die veröffentlichten Schlüssel des Providers, den Issuer gegen den Discovery-Issuer, die Audience gegen Ihre eigene Client-ID, die Gültigkeitsdauer mit ein paar Minuten Toleranz für Uhrabweichungen und die Nonce gegen den Wert, der mit dem ausstehenden Datensatz gespeichert wurde. Die Schlüssel kommen über einen Configuration Manager, der sie aktualisiert, vom JWKS-Endpunkt, sodass ein Schlüsselwechsel beim Provider nicht um drei Uhr morgens zum Ausfall wird.

```

var parameters = new TokenValidationParameters
{
    ValidIssuer      = config.Issuer,          // exakt der Discovery-Issuer
    ValidAudience    = _options.ClientId,     // dieser Client, sonst niemand
    IssuerSigningKeys = config.SigningKeys,    // aus dem JWKS aktualisiert
    ValidateLifetime  = true,
    ClockSkew         = TimeSpan.FromMinutes(2)
};

var principal = new JwtSecurityTokenHandler()
    .ValidateToken(idToken, parameters, out var validated);

if ((string)((JwtSecurityToken)validated).Payload["nonce"] != pending.Nonce)
    throw new SecurityTokenException("Nonce mismatch - sign-in rejected.");

```

Danach kommt das Mapping, und der Bezeichner ist der `sub`-Claim im Paar mit dem Issuer – nie die E-Mail-Adresse, die umbenannt oder einem neuen Mitarbeiter zugewiesen werden kann. `name`, `preferred_username` und `email` dienen nur der Anzeige. Gruppen-Claims aus dem Verzeichnis werden als Beleg festgehalten, aber die Berechtigungen, die das Portal durchsetzt, löst weiterhin sein eigener `PermissionService` in den `SessionContext` auf: Der Provider sagt, wer die Person ist, `AccessOps` entscheidet, was sie hier tun darf.

Passt kein Mitarbeiter, legt die Provisionierung bei der ersten Anmeldung einen an – in derselben Transaktion, die den Login-Audit-Eintrag schreibt, mit einer Platzhalter-Abteilung und ohne erhöhte Rechte. Speichern Sie den Issuer zusammen mit dem Subject, denn Subject-Werte sind nur innerhalb eines Issuers eindeutig, und ein später hinzugefügter zweiter Provider würde sonst mit dem ersten kollidieren.

### Ablauf, Refresh und Abmeldung aus beiden Sessions

Ein ID-Token belegt, dass zu einem bestimmten Zeitpunkt ein Login stattgefunden hat; es ist keine Session und sollte auch nicht wie eine behandelt werden. Session-Lebensdauer und Leerlauf-Timeout von `AccessOps` aus Modul 2 gelten weiterhin. Speichern Sie das Ablaufdatum zusammen mit den Tokens, und wenn ein nachgelagerter Aufruf ein Access-Token braucht, das abgelaufen ist, tauschen Sie das Refresh-Token im Back-Channel ein und ersetzen Sie beide, bevor Sie den Aufruf ausführen. Refresh-Tokens sind eigenständige Credentials: Verschlüsseln Sie sie im Ruhezustand, beschränken Sie sie auf einen Benutzer und einen Client, und wenn der Provider sie rotiert, werten Sie die Wiederverwendung eines ausgemusterten Tokens als Kompromittierungssignal und nicht als erneuten Versuch.

Die Abmeldung ist die Hälfte, die regelmäßig vergessen wird. Den `SessionContext` zu entsorgen und die `Wisej.NET`-Session zu verwerfen, beendet nur die lokale Session; die Provider-Session im selben Browser überlebt, sodass die nächste Anmeldung stillschweigend durchläuft und der Benutzer, der sich an einem gemeinsam genutzten Rechner abgemeldet hat, in Wahrheit noch angemeldet ist. Single Sign-out schickt den Browser zusätzlich zum End-Session-Endpunkt, mit dem ID-Token als Hinweis und einer registrierten `Post-Logout-Redirect-URI`. Behandeln Sie auch die Gegenrichtung, wo Ihr Provider sie unterstützt: Eine Logout-Benachrichtigung vom Provider sollte die zugehörige `AccessOps`-Session beenden.

- Die lokale Abmeldung entsorgt den `SessionContext` und verwirft die Session.
- Der End-Session-Redirect schließt die Provider-Session im selben Browser.

- Der Passwortweg aus Modul 2 bleibt nur für benannte Service-Konten bestehen, mit Rate Limiting und Audit.



### **Sicherheitsgewohnheit**

Dem Browser wird nie vertraut. Jede relevante Entscheidung – wer der Benutzer ist, was er tun darf, welchen Datensatz er sehen darf – wird auf dem Server in einem Service getroffen, und die UI spiegelt sie nur wider. Existiert eine Regel nur als versteckter Button, dann existiert sie nicht.



### **Lernziele**

Am Ende dieses Moduls können Sie:

- ✓ den Authorization Code Flow mit PKCE erklären und wofür jedes Token dient.
- ✓ eine Wisej.NET-Session zu einem Identity Provider umleiten und den Callback verarbeiten, ohne die Session zu verlieren.
- ✓ das ID-Token validieren, Provider-Claims auf den SessionContext abbilden und den lokalen Benutzer bei der ersten Anmeldung provisionieren.
- ✓ Token-Ablauf, Refresh und Single Sign-out behandeln.

## Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab erstellen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



### Praxis-Lab

Ergänzen Sie AccessOps um eine externe Anmeldung: einen OpenID-Connect-Client, der den Authorization Code Flow mit PKCE gegen einen Test-Identity-Provider nutzt, einen Callback-Handler, der das ID-Token und seine Nonce validiert, ein Claim-Mapping, das den SessionContext befüllt und bei der ersten Anmeldung einen lokalen Mitarbeiterdatensatz anlegt, einen Rückfall auf das lokale Login für Service-Konten und eine Abmeldung, die sowohl die lokale als auch die Provider-Session beendet.

## Ausgangspunkt

Öffnen Sie die AccessOps-Solution so, wie Sie sie nach Modul 3 hinterlassen haben: Die Login-Seite prüft einen gesalzenen Passwort-Hash, der SessionContext trägt eine verifizierte Identität, und der PermissionService löst Rollen und Claims in Berechtigungen auf, die jede Methode von DocumentService und ApprovalService prüft. Sie benötigen einen Test-Identity-Provider unter Ihrer Kontrolle – einen lokalen Keycloak-Realm, einen Entwickler-Tenant oder einen beliebigen konformen Server – mit einem für AccessOps registrierten Confidential Client und der Callback-URL Ihrer Entwicklungsumgebung in der Liste der erlaubten Redirect-URLs.

## Vorgeschlagenes Vorgehen

- ✓ Registrieren Sie den AccessOps-Client beim Provider, lesen Sie dann beim Start sein Discovery-Dokument und speichern Sie die Authorize-, Token-, End-Session- und JWKS-Adressen zwischen, statt sie fest zu codieren.
- ✓ Fügen Sie einen Button `Sign in with your work account` hinzu, der die Authorize-URL mit State, Nonce und einer S256-Code-Challenge aufbaut, auf dem Server einen Datensatz für die ausstehende Anmeldung mit dem State als Schlüssel ablegt und `Application.Navigate` aufruft.
- ✓ Schreiben Sie den Callback-Endpunkt als Wisej.NET-HTTP-Handler oder als begleitenden ASP.NET-Core-Endpunkt: State abgleichen, Code und Verifier über einen HTTPS-Aufruf im Back-Channel gegen Tokens eintauschen und Signatur, Issuer, Audience, Gültigkeitsdauer und Nonce des ID-Tokens validieren.
- ✓ Bilden Sie das Paar aus Issuer und Subject auf einen Mitarbeiterdatensatz ab und legen Sie bei der ersten Anmeldung einen ohne erhöhte Berechtigungen an; geben Sie das Ergebnis dann über ein Einmal-Ticket an die Anwendung zurück und bauen Sie daraus den SessionContext auf.
- ✓ Implementieren Sie die Abmeldung in beide Richtungen und erhalten Sie den Passwortweg nur für benannte Service-Konten; demonstrieren Sie anschließend einen abgewiesenen Callback mit manipuliertem State und einem wiederverwendeten ID-Token.

## Geforderte Ergebnisse

- ✓ OIDC-Client-Konfiguration mit PKCE
- ✓ Callback-Verarbeitung mit Token- und Nonce-Validierung

- ✓ Claim-Mapping und Provisionierung bei der ersten Anmeldung
- ✓ Rückfall auf lokales Login für Service-Konten
- ✓ Abmeldung, die beide Sessions abdeckt

## Wiederholungsfragen

- ✓ Ein Angreifer liest den Authorization Code aus dem Browserverlauf eines gemeinsam genutzten Rechners aus. Was hindert ihn daran, ihn gegen Tokens einzutauschen, und was hätte ihn ohne PKCE nicht aufgehalten?
- ✓ Ihr Provider stellt Gruppen-Claims aus, die genau wie die AccessOps-Berechtigungsnamen aussehen. Warum löst der PermissionService die Berechtigungen trotzdem selbst auf, statt diese Claims direkt in den SessionContext zu kopieren?
- ✓ Ein Benutzer klickt auf „Abmelden“, schließt nichts, klickt sofort wieder auf „Anmelden“ – und landet wieder in AccessOps, ohne etwas eingeben zu müssen. Welche Session hat überlebt, und welcher Redirect fehlte?



### Sicherheitsgewohnheit

Dem Browser wird nie vertraut. Jede relevante Entscheidung – wer der Benutzer ist, was er tun darf, welchen Datensatz er sehen darf – wird auf dem Server in einem Service getroffen, und die UI spiegelt sie nur wider. Existiert eine Regel nur als versteckter Button, dann existiert sie nicht.