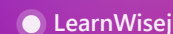


Intégration d'OAuth et d'OpenID Connect

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

Déléguer la connexion à un fournisseur d'identité, le flux authorization code, la gestion des redirections dans une application Wisej.NET, la validation des jetons et le mappage des claims vers le SessionContext. Ce que cela implique pour un développeur qui sécurise une application Wisej.NET de production — et comment aborder ce module.



Enjeux en production

Déléguer la connexion à un fournisseur d'identité, le flux authorization code, la gestion des redirections dans une application Wisej.NET, la validation des jetons et le mappage des claims vers le SessionContext.

1. Le problème en production

AccessOps sait déjà vérifier ses propres mots de passe, mais l'organisation qui l'exploite ne veut pas d'un second annuaire : le personnel se connecte à tout le reste avec le fournisseur d'identité de l'entreprise, et les arrivées, les départs et la politique d'authentification multifacteur sont administrés en un seul endroit. Déléguer la connexion supprime entièrement le mot de passe d'AccessOps et le remplace par une autre obligation : le portail reçoit désormais une identité d'un tiers, par l'intermédiaire du navigateur de l'utilisateur, et doit prouver que cette identité est authentique, non modifiée et bien destinée à lui avant d'afficher le moindre enregistrement de département.

2. Concepts clés

Concept	Pourquoi c'est important
flux authorization code	Le flux OpenID Connect dans lequel le navigateur ne transporte jamais qu'un code à usage unique et de courte durée, tandis que les jetons sont récupérés par votre serveur via un appel HTTPS direct que le navigateur ne voit jamais.
PKCE	Un code verifier aléatoire généré à chaque connexion, envoyé au point de terminaison authorize uniquement sous forme de hachage SHA-256 et révélé au point de terminaison token, de sorte qu'un code intercepté ne peut être échangé par personne d'autre.
ID token	Un JWT signé qui atteste que ce fournisseur a authentifié ce sujet, pour ce client, à ce moment. C'est le seul jeton qu'AccessOps consulte pour décider qui est l'utilisateur.
access token et refresh token	L'access token est un identifiant au porteur permettant d'appeler l'API de quelqu'un d'autre ; le refresh token permet d'obtenir des remplaçants via le canal arrière. Ni l'un ni l'autre ne prouve l'identité pour votre propre application.
state et nonce	Deux valeurs aléatoires que vous générez à chaque connexion : state rattache le callback à la requête que vous avez réellement lancée, nonce rattache l'ID token renvoyé à cette même requête, pour qu'un jeton plus ancien ne puisse pas y être rejoué.
redirect URI	L'URL exacte, préalablement enregistrée, vers laquelle le fournisseur renvoie le navigateur. Dans AccessOps, il s'agit d'un petit endpoint côté serveur plutôt que d'une fenêtre Wisej.NET, car il doit s'exécuter avant qu'aucune UI n'existe.
mappage des claims et provisionnement	Transformer les claims du fournisseur en employé AccessOps : le couple émetteur / sujet identifie le compte, et une ligne locale est créée à la première connexion, sans aucune permission tant que quelqu'un ne les accorde pas.

3. Déroulé de la leçon

Trois jetons et un aller-retour

Tout commence par le document de découverte du fournisseur, l'URL de configuration well-known qui liste les points de terminaison authorize, token et end-session ainsi que l'adresse JWKS de ses clés de signature. AccessOps le lit une fois au démarrage et le met en cache, car chaque étape ultérieure dépend des valeurs exactes qu'il publie — avant tout la chaîne de l'émetteur, qui doit correspondre caractère pour caractère.

La connexion elle-même se compose de deux demi-conversations. Dans le canal avant, le navigateur est envoyé vers le point de terminaison authorize avec l'id client, la redirect URI enregistrée, `response_type=code`, les scopes dont AccessOps a besoin (`openid profile email`), un nouveau state, un nouveau nonce et un code challenge avec `code_challenge_method=S256`. L'utilisateur s'authentifie auprès du fournisseur — AccessOps ne voit jamais cette page et ne voit jamais le mot de passe — et le navigateur revient vers la redirect URI en ne portant qu'un code et le state. Dans le canal arrière, votre

serveur envoie ce code au point de terminaison token avec le code verifier et l'identifiant du client, et reçoit les jetons dans le corps de la réponse.

Chaque jeton a exactement un rôle. L'ID token est le reçu d'authentification et le seul élément qui répond à la question de savoir qui s'est connecté. L'access token est un identifiant destiné à une API de ressources, comme le graphe de l'annuaire, et ne doit jamais être inspecté pour décider de l'identité dans votre propre application ; il peut être opaque, et il est adressé à une autre audience. Le refresh token permet d'obtenir de nouveaux jetons sans repasser par le navigateur. PKCE est ce qui rend ce découpage sûr : sans le verifier, un code volé dans un historique de navigation, un en-tête referrer ou un proxy compromis ne vaut rien pour celui qui le détient.

Quitter la session Wisej.NET et retrouver son chemin

Dans AccessOps, la redirection part du code serveur avec `Application.Navigate`, qui dirige le navigateur vers l'URL autorisée que vous avez construite. Le navigateur a alors quitté votre application, et il reviendra vers la redirect URI — qui ne peut pas être une fenêtre Wisej.NET, puisqu'elle doit répondre à une requête HTTP qui arrive avant qu'aucune fenêtre n'existe. Le callback est ici un petit endpoint côté serveur hébergé dans la même application : un handler HTTP Wisej.NET personnalisé, ou un endpoint ASP.NET Core associé sur la même origine. Son seul travail consiste à échanger le code, à valider et à effectuer une redirection.

Le fil qui ramène au contexte de l'utilisateur, c'est la valeur state. Avant de naviguer, AccessOps crée un identifiant aléatoire et stocke sur le serveur un enregistrement de connexion en attente associé à celui-ci : le code verifier, le nonce, le moment de sa création et la destination que l'utilisateur voulait atteindre. Seul l'identifiant circule dans la query string. Le callback le recherche, rejette tout ce qui est inconnu, expiré ou déjà utilisé, et supprime l'enregistrement pour qu'il ne puisse pas être échangé deux fois. Rien d'autre dans cette query string n'est considéré comme fiable, car tout le canal avant est sous le contrôle du navigateur.

Une fois les jetons validés, le handler redirige vers l'URL de démarrage d'AccessOps avec un ticket à usage unique ; l'application lit ce ticket au démarrage dans la query string de la requête, l'échange une seule fois auprès du même stockage, et construit le SessionContext à partir de l'enregistrement plutôt qu'à partir de ce que le navigateur a transporté. Cette indirection garde la session intègre, que la session d'origine soit toujours active ou que le retour en ait démarré une nouvelle. Ouvrir le fournisseur dans une seconde fenêtre du navigateur et signaler la session en attente avec `Application.Update` est une forme tout aussi valable ; la règle ne change pas.

Valider l'ID token, puis le mapper

Un JWT est signé, pas secret : n'importe qui peut le décoder, et lire les claims avant de valider est donc l'erreur classique. AccessOps vérifie cinq choses avec `System.IdentityModel.Tokens.Jwt` et les packages `Microsoft.IdentityModel` : la signature au regard des clés publiées par le fournisseur, l'émetteur au regard de l'émetteur du document de découverte, l'audience au regard de votre propre id client, la durée de validité avec une tolérance d'horloge de quelques minutes, et le nonce au regard de la valeur stockée avec l'enregistrement en attente. Les clés proviennent du point de terminaison JWKS via un gestionnaire de configuration qui les rafraîchit, pour qu'une rotation de clés chez le fournisseur ne se transforme pas en panne à trois heures du matin.

```

var parameters = new TokenValidationParameters
{
    ValidIssuer      = config.Issuer,          // exactly the discovery issuer
    ValidAudience    = _options.ClientId,     // this client, nobody else
    IssuerSigningKeys = config.SigningKeys,    // refreshed from the JWKS
    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.");

```

Le mappage vient ensuite, et l'identifiant est le claim `sub` associé à l'émetteur — jamais l'adresse e-mail, qui peut être renommée ou réattribuée à un nouvel arrivant. `name`, `preferred_username` et `email` servent uniquement à l'affichage. Les claims de groupe issus de l'annuaire sont enregistrés à titre de preuve, mais les permissions que le portail fait respecter sont toujours résolues par son propre `PermissionService` dans le `SessionContext` : le fournisseur dit qui est la personne, `AccessOps` décide de ce qu'elle peut faire ici.

Lorsqu'aucun employé ne correspond, le provisionnement à la première connexion en crée un dans la même transaction que celle qui écrit l'enregistrement d'audit de connexion, avec un département provisoire et aucun droit élevé. Stockez l'émetteur avec le sujet, car les valeurs de sujet ne sont uniques qu'au sein d'un même émetteur, et un second fournisseur ajouté plus tard entrerait sinon en collision avec le premier.

Expiration, rafraîchissement et déconnexion des deux sessions

Un ID token prouve qu'une connexion a eu lieu à un instant donné ; ce n'est pas une session et il ne doit pas être traité comme tel. La durée de vie de session et le délai d'inactivité d'`AccessOps` définis au Module 2 continuent de s'appliquer. Conservez la date d'expiration des jetons avec les jetons et, lorsqu'un appel en aval a besoin d'un access token expiré, échangez le refresh token via le canal arrière et remplacez les deux avant d'effectuer l'appel. Les refresh tokens sont des identifiants à part entière : chiffrez-les au repos, limitez-les à un utilisateur et à un client et, lorsque le fournisseur les fait tourner, traitez la réutilisation d'un jeton retiré comme un signal de compromission plutôt que comme une nouvelle tentative.

La déconnexion est la moitié que l'on oublie régulièrement. Libérer le `SessionContext` et abandonner la session `Wisej.NET` ne met fin qu'à la session locale ; la session du fournisseur dans le même navigateur survit, si bien que la connexion suivante aboutit silencieusement et que l'utilisateur qui s'est déconnecté sur un poste partagé ne l'est pas vraiment. La déconnexion unique envoie aussi le navigateur vers le point de terminaison end-session avec l'ID token comme indice et une post-logout redirect URI enregistrée. Gérez également l'autre sens lorsque votre fournisseur le permet : une notification de déconnexion envoyée par le fournisseur doit mettre fin à la session `AccessOps` correspondante.

- La déconnexion locale libère le `SessionContext` et abandonne la session.
- La redirection end-session ferme la session du fournisseur dans le même navigateur.

- Le chemin par mot de passe du Module 2 ne subsiste que pour des comptes de service nommés, avec limitation du débit et audit.



Réflexe sécurité

Le navigateur n'est jamais digne de confiance. Chaque décision importante — qui est l'utilisateur, ce qu'il peut faire, quel enregistrement il peut voir — est prise sur le serveur, dans un service, et l'UI se contente de la refléter. Si une règle n'existe que sous la forme d'un bouton masqué, elle n'existe pas.



Objectifs d'apprentissage

À la fin de ce module, vous serez capable de :

- ✓ Expliquer le flux authorization code avec PKCE et le rôle de chaque jeton.
- ✓ Rediriger une session Wisej.NET vers un fournisseur d'identité et traiter le callback sans perdre la session.
- ✓ Valider l'ID token, mapper les claims du fournisseur vers le SessionContext et provisionner l'utilisateur local à la première connexion.
- ✓ Gérer l'expiration des jetons, leur rafraîchissement et la déconnexion unique.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.



Lab pratique

Ajoutez la connexion externe à AccessOps : un client OpenID Connect qui utilise le flux authorization code avec PKCE face à un fournisseur d'identité de test, un handler de callback qui valide l'ID token et son nonce, un mappage des claims qui remplit le SessionContext et crée un enregistrement d'employé local à la première connexion, un repli sur la connexion locale pour les comptes de service, et une déconnexion qui met fin à la fois à la session locale et à celle du fournisseur.

Point de départ

Ouvrez la solution AccessOps telle que vous l'avez laissée après le Module 3 : la page de connexion vérifie un hachage de mot de passe salé, le SessionContext porte une identité vérifiée, et le PermissionService résout les rôles et les claims en permissions contrôlées par chaque méthode de DocumentService et d'ApprovalService. Il vous faut un fournisseur d'identité de test que vous maîtrisez — un realm Keycloak local, un tenant de développement ou tout serveur conforme — avec un client confidentiel enregistré pour AccessOps et l'URL de callback de votre site de développement ajoutée à ses redirect URIs autorisées.

Approche suggérée

- ✓ Enregistrez le client AccessOps auprès du fournisseur, puis lisez son document de découverte au démarrage et mettez en cache les adresses authorize, token, end-session et JWKS au lieu de les coder en dur.
- ✓ Ajoutez un bouton `Sign in with your work account` qui construit l'URL authorize avec state, nonce et un code challenge S256, dépose sur le serveur un enregistrement de connexion en attente indexé par state, et appelle `Application.Navigate`.
- ✓ Écrivez l'endpoint de callback sous forme de handler HTTP Wisej.NET ou d'endpoint ASP.NET Core associé : faites correspondre le state, échangez le code et le vérifier contre des jetons via un appel HTTPS en canal arrière, et validez la signature, l'émetteur, l'audience, la durée de validité et le nonce de l'ID token.
- ✓ Mappez le couple émetteur / sujet vers un enregistrement d'employé, en créant un sans permission élevée à la première connexion, puis renvoyez le résultat à l'application via un ticket à usage unique et construisez le SessionContext à partir de celui-ci.
- ✓ Mettez en œuvre la déconnexion dans les deux sens et ne conservez le chemin par mot de passe que pour des comptes de service nommés, puis démontrez un callback rejeté avec un state falsifié et un ID token rejoué.

Livrables attendus

- ✓ Configuration du client OIDC avec PKCE
- ✓ Gestion du callback avec validation du jeton et du nonce
- ✓ Mappage des claims et provisionnement à la première connexion

- ✓ Repli sur la connexion locale pour les comptes de service
- ✓ Déconnexion couvrant les deux sessions

Questions de révision

- ✓ Un attaquant récupère le code d'autorisation dans l'historique d'un navigateur sur un poste partagé. Qu'est-ce qui l'empêche de l'échanger contre des jetons, et qu'est-ce qui n'aurait pas suffi à l'en empêcher sans PKCE ?
- ✓ Votre fournisseur émet des claims de groupe qui ressemblent exactement aux noms des permissions d'AccessOps. Pourquoi le PermissionService résout-il malgré tout les permissions lui-même au lieu de copier ces claims directement dans le SessionContext ?
- ✓ Un utilisateur clique sur Sign out, ne ferme rien, puis clique immédiatement sur Sign in — et se retrouve dans AccessOps sans rien saisir. Quelle session a survécu, et quelle redirection manquait ?



Réflexe sécurité

Le navigateur n'est jamais digne de confiance. Chaque décision importante — qui est l'utilisateur, ce qu'il peut faire, quel enregistrement il peut voir — est prise sur le serveur, dans un service, et l'UI se contente de la refléter. Si une règle n'existe que sous la forme d'un bouton masqué, elle n'existe pas.