MODÈLES DE SÉCURITÉ ET D'AUTHENTIFICATION · MODULE 4

Connexion personnalisée et sécurité de session

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

LearnWisej

Guide de la leçon

Maîtriser la transaction de connexion, établir le principal applicatif et faire tourner l'ID de session, n'émettre les cookies de l'hôte que par HTTP, traiter la déconnexion comme la révocation d'un octroi côté serveur, et fermer les chemins de falsification de requête, d'origine WebSocket et de redirection. Ce que cela signifie pour un développeur qui sécurise les chemins de connexion, d'identité et d'autorisation d'une application Wisej.NET en production — et comment aborder ce module.

Objectif en production

Maîtriser la transaction de connexion, établir le principal applicatif et faire tourner l'ID de session, n'émettre les cookies de l'hôte que par HTTP, traiter la déconnexion comme la révocation d'un octroi côté serveur, et fermer les chemins de falsification de requête, d'origine WebSocket et de redirection.

1. Le problème en production

Les partenaires atteignent AccessOps par la route HTTPS publique, où il n'y a aucun challenge Windows sur lequel s'appuyer : l'application maîtrise toute la connexion, de la zone du mot de passe jusqu'au moment où la console remplace la vue de connexion. Chaque raccourci devient un chemin d'attaque : un verrouillage qui révèle quels noms existent, un ID de session qui survit à la connexion, ou une déconnexion qui masque un bouton pendant qu'un second onglet continue d'exporter. Ce module construit la connexion des partenaires et prouve qu'un octroi révoqué est refusé partout.

2. Concepts clés

Concept	Pourquoi c'est important
Transaction de connexion	La séquence côté serveur limitation → vérification → résolution du compte et du tenant → application de la politique → création de l'octroi → établissement du principal, où toute étape en échec met fin à la tentative sans laisser d'état derrière elle.
Limitation des connexions	Des compteurs des échecs récents par compte et par adresse cliente qui refusent toute nouvelle tentative pendant une période de verrouillage, tenus dans le service de connexion, car un clic Wisej ne peut pas être limité par un middleware ASP.NET Core.
Échec générique	Une seule phrase identique pour un mauvais mot de passe, un nom inconnu, un compte désactivé et un verrouillage, avec un temps de réponse comparable, pour que l'écran ne confirme jamais qu'un compte existe.
Principal applicatif	Le `ClaimsPrincipal` affecté à `Application.User` pour cette session Wisej ; c'est un état de session, pas un cookie de l'hôte ni un jeton Windows.
Rotation de l'ID de session	`Application.RefreshSessionId()` donne à la session un nouvel ID sans perdre son état, si bien qu'un ID qui existait avant la connexion ne vaut plus rien après.
Cookie d'authentification de l'hôte	Un ticket de cookie ASP.NET Core émis avec `Secure` et `HttpOnly` par un endpoint HTTPS, nécessaire seulement lorsque des endpoints HTTP ordinaires doivent reconnaître l'utilisateur.
Octroi de connexion	Un enregistrement côté serveur avec un ID aléatoire opaque, une clé de compte, un tenant, une expiration et un indicateur de révocation, vérifié par chaque opération protégée ; le révoquer, c'est cela, se déconnecter.

3. Le fil de la leçon

Maîtriser la transaction de connexion

La vue de connexion d'AccessOps sur la route publique anonyme fait exactement une chose dans `btnSignIn_Click` : elle lit les deux zones, vide la zone du mot de passe et appelle `LoginService.SignIn`. La vérification des identifiants revient à un système d'identité pris en charge — ASP.NET Core Identity, un annuaire ou, pour les comptes locaux, une dérivation fournie par le framework comme PBKDF2 via `Rfc2898DeriveBytes.Pbkdf2`, avec un sel par compte et une comparaison en temps constant. Quand le nom d'utilisateur est inconnu, le service vérifie quand même un identifiant factice, pour que « utilisateur inexistant » coûte le même temps que « mauvais mot de passe ».

La limitation vit dans ce même service. Le `LoginThrottle` d'AccessOps verrouille un compte pendant 15 minutes après cinq échecs dans une fenêtre de 15 minutes, et une adresse après vingt, car le credential stuffing essaie de nombreux noms depuis un même endroit. La limitation de débit d'ASP.NET Core fonctionne sur les endpoints placés à côté d'`app.UseWisej()`, mais ne voit pas un clic Wisej, qui arrive par le WebSocket de la session. Lisez l'adresse depuis `Application.UserHostAddress` et gardez ses limites en tête : un même navigateur peut apparaître à la fois comme `::1` et `127.0.0.1`, et derrière un proxy, le

vrai client ne provient que des en-têtes transférés auxquels vous faites confiance. Des compteurs en mémoire du processus ne sont corrects que pour un seul serveur.

Chaque refus — mauvais mot de passe, nom inconnu, compte désactivé, verrouillage — affiche une seule phrase : « That user name and password combination is not valid. » La vraie raison ne va que dans la table d'audit des connexions. Le mot de passe lui-même ne va nulle part : ni dans `Application.Session`, ni dans un message de journal, ni dans le texte d'une exception.

N'établir le principal qu'une fois la politique satisfaite

Un mot de passe vérifié prouve la connaissance d'un secret, pas le droit d'utiliser AccessOps. Le service résout le compte local et son tenant, refuse un compte désactivé ou un tenant non approuvé avec la même phrase générique, crée un octroi de connexion, et ce n'est qu'ensuite qu'il construit le principal. L'utilitaire `ApplicationPrincipal.Establish` du pack transforme compte, tenant, octroi et rôles en claims et affecte `Application.User = new ClaimsPrincipal(new ClaimsIdentity(claims, "Portal"))`. Ce principal est un état de session : il n'émet aucun cookie, le middleware ASP.NET Core ne le voit jamais sur les autres requêtes, et il ne crée aucun jeton Windows.

L'ordre compte et, sur Wisej.NET 4.1.7, le type de requête aussi. Appeler `Application.RefreshSessionId()` depuis un clic — un événement WebSocket — change l'ID sur le serveur, mais le navigateur ne l'adopte jamais ; sa requête suivante démarre une toute nouvelle session et la connexion est perdue. Appelée pendant un chargement de page, la méthode fait adopter le nouvel ID par le navigateur, qui le conserve dans le sessionStorage de l'onglet. AccessOps vérifie donc au clic, met le résultat vérifié en attente sous une clé, appelle `Application.Reload()`, et termine au chargement :

```
// Program.Main: Application.ApplicationRefresh += (s, e) => OnPageLoad();
private static void OnPageLoad()
{
    if (Application.Session["PendingSignIn"] is not VerifiedSignIn p)
        return;

    Application.Session["PendingSignIn"] = null;
    Application.RefreshSessionId();          // new ID first (HTTP load)
    ApplicationPrincipal.Establish(p.AccountId, p.TenantId, p.GrantId, p.Roles);

    var login = Application.MainPage;
    Application.MainPage = new ConsolePage(); // replace the login view
    login?.Dispose();
}
```

La copie d'`ApplicationPrincipal` du cours affecte donc `Application.User` sans faire tourner l'ID ; le chargement le fait tourner une fois, avant que l'identité n'existe.

Les cookies par HTTP, la déconnexion comme révocation

Une trame WebSocket n'a pas d'en-têtes de réponse ; un clic ne peut donc pas livrer de `Set-Cookie` avec `HttpOnly`. `Application.Cookies` ne peut écrire de cookies HttpOnly que pendant `Program.Main`, `ApplicationStart`, `ApplicationRefresh` ou quand `Application.IsWebSocket` vaut false, et `Application.Browser.CookieStorage` est écrit par JavaScript, qui ne peut jamais définir HttpOnly. Quand

des endpoints HTTP ordinaires doivent reconnaître le partenaire, AccessOps mappe avant `app.UseWisej()` un endpoint HTTPS qui échange un handle à usage unique et de courte durée créé par la session, appelle `HttpContext.SignInAsync` avec le schéma `"Cookies"`, et laisse le handler de cookies émettre `Secure`, `HttpOnly` et `SameSite=Strict`. Ce ticket n'appartient pas à Wisej : l'ID de session vit dans le sessionStorage, et Wisej 4.1.7 écrit son propre cookie HttpOnly de validation des requêtes, auquel vous ne touchez pas.

La déconnexion n'est ni un bouton masqué ni un cookie supprimé. Le `GrantStore` contient des octrois opaques et à durée limitée ; `ExportService` vérifie l'octroi désigné par le claim `grant` du principal, le tenant et l'enregistrement avant que le repository ne s'exécute. La déconnexion révoque d'abord l'octroi, écrit l'audit, fait expirer le cookie de l'hôte via l'endpoint de déconnexion, efface l'identité de la session et appelle `Application.Exit()`. Mettre `Application.User` à `null` ne suffit pas — le getter se replie sur le principal de l'hôte et, sous l'authentification Windows d'IIS, un rechargement reconnecte aussitôt l'utilisateur. Un second onglet affiche encore Export ; son clic suivant est refusé à zéro lecture du repository (fixture F05). `CookieAuthenticationEvents.OnValidatePrincipal` applique la même vérification d'octroi aux requêtes de l'hôte.

Fermer les chemins intersites

Le navigateur joint le cookie de l'hôte à n'importe quelle requête, y compris une requête lancée par un autre site ; chaque POST modifiant l'état qui s'y fie — connexion, déconnexion, transfert du cookie — valide donc un jeton antifalsification. L'ID de session Wisej se trouve dans le sessionStorage, qu'une page étrangère ne peut pas faire envoyer par le navigateur, mais la négociation WebSocket transporte quand même les cookies et n'est pas couverte par CORS : un petit middleware placé avant `app.UseWisej()` refuse un upgrade dont l'`Origin` n'est pas une origine AccessOps, et l'ingress répète la règle.

Les URL de continuation sont des entrées. Un `returnUrl` issu d'`Application.QueryString` ou de la query d'un endpoint n'est accepté que s'il s'agit d'un chemin local qui ne commence ni par `//` ni par `/\`, ou d'une entrée d'une liste d'autorisation explicite côté serveur ; tout le reste devient `/`. Ne déduisez le schéma et l'hôte qu'une fois `app.UseForwardedHeaders()` configuré pour des proxys connus. La fixture F11 — un POST ou un upgrade d'une autre origine — doit se terminer sans aucun changement d'état.

> **Réflexe de sécurité**
>
> Gardez les identités séparées et décidez sur le serveur. Le principal de l'hôte, le principal applicatif Wisej.NET et le jeton Windows effectif sont trois faits différents, et aucun n'est une permission. Chaque lecture protégée vérifie l'octroi actuel, le tenant et l'enregistrement à la couche service avant que le repository ne s'exécute, et un bouton masqué, un claim de rôle ou un onglet déconnecté ne prouvent rien tant que cette vérification n'a pas refusé.

> **Objectifs d'apprentissage**
>
> À la fin de ce module, vous saurez :

- ✓ Implémenter une connexion qui vérifie les identifiants avec un système d'identité pris en charge, limite les tentatives, renvoie des échecs génériques et tient les mots de passe à l'écart des journaux et de l'état de session.

- ✓ N’établir le principal applicatif Wisej.NET qu’une fois satisfaites les politiques de compte et de tenant, faire tourner l’ID de session avec Application.RefreshSessionId et remplacer la vue de connexion.
- ✓ Émettre et faire expirer des cookies de l’hôte Secure et HttpOnly depuis un endpoint HTTPS dédié plutôt que depuis un événement WebSocket, et traiter la déconnexion comme la révocation d’un octroi côté serveur vérifié par chaque opération protégée.
- ✓ Fermer les chemins intersites avec un jeton antifalsification sur les POST modifiant l’état, des vérifications d’origine WebSocket et une liste d’autorisation pour les URL de continuation.

Guide du lab / de l'examen

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

Lab pratique

Donnez à la route partenaires d'AccessOps sa propre connexion : une vue de connexion sur la route publique anonyme qui vérifie une identité de fixture par un vérificateur pris en charge ou par une simulation d'entraînement qui n'est jamais livrée, rejette un mauvais identifiant et un compte désactivé avec le même message générique, résout le compte local et le tenant, établit le principal avec ApplicationPrincipal et fait tourner l'ID de session. Appuyez-la sur un GrantStore d'octrois de connexion opaques, à durée limitée et révocables, que le service d'export vérifie avant chaque lecture, ajoutez un endpoint HTTPS de connexion et de déconnexion pour le cookie de l'hôte, puis ouvrez deux profils de navigateur, révoquez l'octroi pendant que le second onglet est ouvert, réessayez l'export dans les deux onglets et après un rafraîchissement, et testez un POST forgé, un upgrade WebSocket d'une autre origine et une URL de continuation externe.

Point de départ

Ouvrez la solution AccessOps du Module 1 avec les fixtures des tenants A et B, le repository compteur de lectures et la garde `DocumentService`, et ajoutez une `LoginPage` sur la route publique anonyme. Exécutez-la sur un nom d'hôte HTTPS (ou `https://localhost`) et gardez deux profils de navigateur prêts pour le test de révocation.

Approche suggérée

- ✓ Écrivez `LoginService.SignIn` avec `LoginThrottle`, une vérification d'identifiant factice pour les noms inconnus et une seule phrase d'échec générique ; consignez les résultats dans la table d'audit et confirmez qu'aucune valeur de mot de passe n'atteint un journal ni `Application.Session`.
- ✓ Ajoutez `GrantStore` (ID aléatoire opaque, compte, tenant, expiration, indicateur de révocation) et faites vérifier par `ExportService` l'octroi, le tenant et l'enregistrement avant que le repository ne s'exécute.
- ✓ Mettez la connexion vérifiée en attente au clic, appelez `Application.Reload()`, puis, dans `ApplicationRefresh`, appelez `Application.RefreshSessionId()`, puis `ApplicationPrincipal.Establish`, puis remplacez la vue de connexion par la console ; journalisez l'ancien et le nouvel ID sous la forme d'un booléen « rotated ».
- ✓ Mappez avant `app.UseWisej()` les endpoints HTTPS de transfert du cookie et de déconnexion, avec antifalsification, une vérification de l'`Origin` sur les upgrades WebSocket et une vérification de chemin local pour `returnUrl` ; capturez les attributs de `Set-Cookie` avec la valeur masquée.
- ✓ Révoquez l'octroi depuis l'onglet 1, réessayez Export dans l'onglet 2 et après un rafraîchissement, puis exécutez les tests de POST forgé, d'upgrade d'une mauvaise origine et d'URL de continuation externe, et dessinez le diagramme des transitions de l'état d'authentification.

Livrables attendus

- ✓ Vue de connexion avec échecs génériques, limitation et gestion des comptes désactivés
- ✓ Établissement du principal avec rotation de l'ID de session
- ✓ Endpoint HTTPS du cookie avec attributs Set-Cookie masqués
- ✓ GrantStore avec révocation appliquée à tous les onglets ouverts
- ✓ Diagramme des transitions de l'état d'authentification et tests de POST forgé, de mauvaise origine et d'URL de continuation

Questions de révision

- ✓ Votre premier jet appelait `Application.RefreshSessionId()` dans `btnSignIn_Click` : la console est apparue, puis a disparu au clic suivant. Qu'a fait le navigateur du nouvel ID, et pourquoi un chargement de page se comporte-t-il différemment ?
- ✓ Après la déconnexion, l'onglet 2 affiche toujours le bouton Export et détient toujours un principal avec un claim de rôle. Quelle vérification unique fait échouer son export suivant, et pourquoi supprimer le seul cookie de l'hôte n'y parviendrait-il pas ?
- ✓ L'ID de session Wisej n'est pas dans un cookie, et pourtant AccessOps a encore besoin d'antifalsification et d'une vérification d'origine. Quelles requêtes portent des identifiants ambiants qu'un autre site pourrait déclencher ?

Réflexe de sécurité

Gardez les identités séparées et décidez sur le serveur. Le principal de l'hôte, le principal applicatif Wisej.NET et le jeton Windows effectif sont trois faits différents, et aucun n'est une permission. Chaque lecture protégée vérifie l'octroi actuel, le tenant et l'enregistrement à la couche service avant que le repository ne s'exécute, et un bouton masqué, un claim de rôle ou un onglet déconnecté ne prouvent rien tant que cette vérification n'a pas refusé.