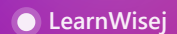


Authentification et identité de session

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



Guide de la leçon

Flux de connexion, gestion des mots de passe, un SessionContext portant une identité vérifiée, délais d'expiration, déconnexion, « se souvenir de moi » et fixation de session. Ce que cela implique pour un développeur qui sécurise une application Wisej.NET de production — et comment aborder ce module.



Enjeux en production

Flux de connexion, gestion des mots de passe, un SessionContext portant une identité vérifiée, délais d'expiration, déconnexion, « se souvenir de moi » et fixation de session.

1. Le problème en production

AccessOps contient des dossiers d'employés, des documents confidentiels et des demandes d'approbation : le premier écran que l'on atteint est donc aussi le premier contrôle qui compte. Tout ce que le reste de ce cours fait respecter — permissions, règles au niveau des enregistrements, pistes d'audit — repose sur une seule hypothèse : le serveur sait qui se trouve à l'autre bout de cette session et peut le prouver. Le Module 1 a tracé les frontières de confiance ; ce module place une identité vérifiée à l'intérieur de la frontière de session et l'empêche d'être devinée, empruntée ou prolongée au-delà de sa durée de vie.

2. Concepts clés

Concept	Pourquoi c'est important
hachage lent et salé	Une dérivation à sens unique du mot de passe, avec un sel aléatoire propre à chaque utilisateur et un nombre d'itérations volontairement coûteux, de sorte qu'une table volée ne peut pas être inversée et que chaque tentative coûte un vrai travail.
comparaison à temps constant	Comparer deux hachages en un nombre fixe d'opérations avec <code>CryptographicOperations.FixedTimeEquals</code> , afin que le temps de réponse du serveur ne révèle jamais combien d'octets d'une tentative étaient justes.
SessionContext	L'objet créé uniquement après la vérification d'un mot de passe, qui contient l'id de l'employé, son département, ses rôles et les détails de sa connexion, et qui est stocké dans <code>Application.Session</code> comme l'unique endroit où le code demande qui est l'utilisateur.
expiration de session	La période d'inactivité au terme de laquelle Wisej.NET libère la session côté serveur et tout ce qu'elle contient, configurée pour l'application dans <code>Default.json</code> plutôt que devinée écran par écran.
fixation de session	Une attaque dans laquelle l'identifiant qui portait la page de connexion anonyme porte ensuite aussi la session authentifiée, si bien qu'un identifiant planté par l'attaquant devient un identifiant connecté.
limitation des tentatives de connexion	Compter les échecs par compte et par adresse client et refuser les tentatives suivantes pendant une période de verrouillage : c'est ce qui transforme une machine à deviner illimitée en quelques essais par heure.
enregistrement d'audit de connexion	Une ligne par tentative avec le nom d'utilisateur essayé, l'heure, l'adresse client et le résultat réel — l'endroit où la raison précise de l'échec est consignée, puisque l'écran n'en affiche jamais qu'une version générique.

3. Déroulé de la leçon

Vérifier un mot de passe que vous ne conservez jamais

AccessOps ne stocke aucun mot de passe. Pour chaque employé, il stocke trois valeurs : l'algorithme et le nombre d'itérations utilisés, un sel aléatoire issu d'un générateur cryptographique, et les octets dérivés. La connexion applique la même dérivation au mot de passe qui vient d'être saisi, avec le sel et le nombre d'itérations de cet employé, et compare le résultat aux octets stockés. Rien n'est déchiffré, car rien n'a été chiffré — un hachage ne fonctionne que dans un sens, et le sel fait que deux employés ayant choisi le même mot de passe produisent des valeurs stockées complètement différentes.

Deux implémentations conviennent ici. PBKDF2 via `Rfc2898DeriveBytes` offre un contrôle explicite : SHA-256, un sel de 16 octets, une sortie de 32 octets et un nombre d'itérations assez élevé pour qu'une vérification prenne une fraction de seconde perceptible. Le `PasswordHasher` d'ASP.NET Core Identity regroupe quant à lui le marqueur d'algorithme, le nombre d'itérations, le sel et le hachage dans une seule chaîne, et renvoie un `PasswordVerificationResult` dont la valeur `SuccessRehashNeeded` vous indique

de refaire la dérivation avec les paramètres actuels. Dans les deux cas, le coût est justement le but : chaque tentative d'un attaquant doit payer une dérivation complète.

Le mot de passe lui-même ne vit que le temps d'un appel de méthode. Il est lu dans la `TextBox` de connexion, passé à `PasswordService.Verify`, puis oublié : jamais un champ de la fenêtre, jamais une valeur de la ligne d'audit, jamais une partie d'un message d'exception, et jamais une ligne de log — pas même tronqué.

```
public bool Verify(string password, StoredCredential stored)
{
    var candidate = Rfc2898DeriveBytes.Pbkdf2(
        Encoding.UTF8.GetBytes(password),
        stored.Salt,
        stored.Iterations,
        HashAlgorithmName.SHA256,
        stored.Hash.Length);

    return CryptographicOperations.FixedTimeEquals(candidate, stored.Hash);
}
```

Une identité vérifiée, et une seule

Lorsque `Verify` renvoie true, le service de connexion construit un `SessionContext` : l'id de l'employé, le nom affiché, l'id du département, l'indication que cette personne encadre ou non d'autres employés, les rôles lus en base à cet instant, l'heure de connexion et l'adresse client. Il est stocké une seule fois dans `Application.Session`, qui est propre à chaque session de navigateur, de sorte que rien ne passe entre la responsable des opérations et le gestionnaire de paie. Un champ statique serait partagé par toutes les sessions du processus.

À partir de là, il est l'unique source de vérité. Les écrans ne gardent aucune copie de l'id utilisateur, et les services ne reçoivent aucun id d'appelant en paramètre, car un paramètre est choisi par le code appelant — `DocumentService` lit lui-même le contexte. Quand la grille des documents affiche un département, ce département provient du contexte et non d'une liste déroulante que l'on aurait pu demander au navigateur de modifier. C'est le réflexe du cours traduit en code : une règle qui n'existe que dans ce que l'UI propose n'est pas une règle.

Le contexte est créé après la vérification, jamais avant. Il n'existe pas de contexte anonyme avec un id vide en attente d'être rempli, car un objet qui existe dans les deux états finira par être considéré comme fiable dans le mauvais. Chaque écran d'AccessOps, à l'exception de la connexion, exige le contexte et renvoie le navigateur en arrière lorsqu'il est absent, et un accesseur typé qui lève une exception quand la session ne contient rien en fait une habitude plutôt qu'une remarque de revue.

Expiration, déconnexion pour inactivité et un Logout qui fait vraiment son travail

Une session Wisej.NET est un état serveur, et sa durée de vie relève de la configuration, pas du code. Le délai d'expiration de session se règle dans le `Default.json` de l'application et, lorsqu'il est écoulé, le serveur libère la session avec les fenêtres, les contrôles et le `SessionContext` qu'elle contient. Choisissez une valeur que le métier peut justifier — vingt minutes conviennent à un portail qui affiche des fourchettes

de salaire et des approbations signées — et gardez-la plus courte que tout délai d'expiration de proxy placé devant l'application, pour que le comportement observé soit bien celui que vous avez configuré.

La déconnexion pour inactivité est la moitié visible de la même règle. Un `Wisej.Web.Timer` relancé à chaque activité de l'utilisateur prévient une minute avant l'échéance, puis exécute le même nettoyage que le bouton Logout : effacer le contexte, supprimer tout ce qui est mis en cache sur la personne, écrire l'enregistrement d'audit et appeler `Application.Exit()`. Les deux chemins appellent une même méthode, car deux routines de nettoyage finissent par diverger, et celle sur laquelle personne ne clique est celle qui oublie quelque chose.

- La session se termine sur le serveur — fermer l'onglet du navigateur n'est pas une déconnexion.
- Tout état d'écran qui désignait une personne a disparu avant que la session ne soit libérée.
- Le navigateur arrive sur la page de connexion, et revenir en arrière dans l'historique n'affiche aucune donnée.

Force brute, credential stuffing et fixation

Une page de connexion sans limitation est une machine à deviner illimitée. `LoginThrottle` tient des compteurs par compte et par adresse client ; après cinq échecs dans une courte fenêtre, le compte est refusé pendant une période de verrouillage, et les compteurs sont remis à zéro en cas de succès. Donnez aussi à chaque réponse d'échec la même forme et pas seulement la même formulation : lorsque le nom d'utilisateur n'existe pas, effectuez quand même une vérification contre un identifiant factice fixe, pour qu'un compte inexistant et un mauvais mot de passe coûtent le même temps.

Le message affiché est une seule phrase générique pour tous les échecs — utilisateur inconnu, mauvais mot de passe, compte verrouillé ou désactivé. La vraie raison a sa place dans la ligne d'audit, où un administrateur peut voir trente tentatives d'utilisateur inconnu depuis une même adresse en dix minutes. Ce schéma, c'est le credential stuffing : des couples identifiant / mot de passe récupérés lors de la fuite de données de quelqu'un d'autre et essayés en masse, stoppés par le compteur par adresse plutôt que par le compteur par compte, puisque chaque compte n'est parfois essayé qu'une fois.

La fixation de session est plus subtile. L'identifiant qui portait la page de connexion anonyme ne doit pas porter l'identité vérifiée : la session d'avant la connexion se termine donc après la vérification et le navigateur continue sur une nouvelle session, sans rien recopier. La fonction « se souvenir de moi » suit la même discipline : jamais un mot de passe stocké ni une valeur devinable, mais un long jeton aléatoire par appareil, haché au repos, lié à un seul employé, doté d'une date d'expiration, révocable depuis la console d'administration, et revérifié contre le compte actif avant que le `SessionContext` ne soit reconstruit.



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 :

- ✓ Mettre en œuvre une page de connexion qui vérifie les identifiants par rapport à un hachage salé et ne stocke jamais le mot de passe.
- ✓ Créer un SessionContext qui conserve une identité vérifiée dans Application.Session et constitue l'unique source de vérité sur l'identité de l'utilisateur.
- ✓ Configurer l'expiration de session, la déconnexion pour inactivité et la déconnexion explicite afin que l'état soit libéré et ne puisse pas être réutilisé.
- ✓ Protéger le flux de connexion contre la force brute, le credential stuffing et la fixation de session.

Guide du lab / de l'examen

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



Lab pratique

Mettez en place l'authentification d'AccessOps : une page de connexion avec verrouillage après cinq échecs, une vérification des mots de passe avec un hachage moderne salé, un `SessionContext` créé uniquement après la vérification et stocké dans la session, un délai d'inactivité qui renvoie l'utilisateur à la connexion et efface l'état, une déconnexion explicite qui met fin à la session, et un enregistrement d'audit pour chaque tentative de connexion.

Point de départ

Vous ouvrez la solution AccessOps créée au Module 1, qui n'est encore qu'un shell : une fenêtre de démarrage, une page de connexion provisoire dont le bouton ouvre simplement la console principale, et le document de modèle de menaces qui désignait l'authentification comme le premier contrôle à construire. La table des employés existe, avec les noms et les départements mais sans identifiants, et rien dans la solution ne sait encore qui est connecté.

Approche suggérée

- ✓ Écrivez d'abord `PasswordService`, avec une méthode `Hash` qui génère un nouveau sel aléatoire et une méthode `Verify` qui refait la dérivation avec le sel et le nombre d'itérations stockés, puis compare avec un test d'égalité à temps constant ; ajoutez les colonnes du sel, du nombre d'itérations et du hachage à la table des employés et créez deux comptes de test via le service.
- ✓ Définissez `SessionContext` avec l'id de l'employé, le nom affiché, l'id du département, l'indicateur de manager, les rôles et l'heure de connexion, ainsi qu'un accesseur typé sur `Application.Session` qui renvoie le contexte courant et lève une exception lorsque personne n'est connecté.
- ✓ Construisez `LoginService.SignIn` comme l'unique chemin qui vérifie un identifiant : demandez à `LoginThrottle` si le compte est verrouillé, vérifiez le mot de passe, écrivez l'enregistrement d'audit quel que soit le résultat et, en cas de succès, remettez les compteurs à zéro, terminez la session d'avant la connexion et créez le `SessionContext` pour la nouvelle.
- ✓ Refaites la page de connexion pour que le gestionnaire de son bouton se contente d'appeler `LoginService` et de réagir au résultat — une seule phrase générique pour tous les échecs, la console en cas de succès — et faites en sorte que tous les autres écrans exigent le contexte et redirigent vers la connexion lorsqu'il est absent.
- ✓ Réglez le délai d'expiration de session dans `Default.json`, ajoutez le minuteur d'inactivité avec son avertissement une minute avant, et écrivez une seule routine `Logout` qui efface le contexte, consigne la déconnexion et appelle `Application.Exit()`, appelée à la fois par le bouton et par le chemin d'inactivité.

Livrables attendus

- ✓ Page de connexion avec verrouillage et messages d'erreur génériques

- ✓ Service de hachage et de vérification des mots de passe
- ✓ SessionContext avec identité vérifiée
- ✓ Délai d'inactivité et déconnexion explicite
- ✓ Enregistrements d'audit des connexions

Questions de révision

- ✓ Votre page de connexion répond à un mauvais mot de passe en 480 ms et à un nom d'utilisateur inconnu en 6 ms. Expliquez ce qu'un attaquant apprend de cette seule différence, et décrivez les deux modifications qui suppriment ce signal sans changer le message.
- ✓ Un collègue propose de conserver l'employé connecté dans une propriété statique, parce que « c'est plus simple que d'aller chercher dans la session à chaque fois ». Décrivez précisément ce que voit le deuxième utilisateur simultané, et à quel moment les deux sessions interfèrent pour la première fois.
- ✓ Votre bouton Logout efface le SessionContext mais ne met pas fin à la session. Nommez deux éléments encore récupérables ensuite, et dites lequel vous jugez le plus grave dans un portail qui affiche des documents confidentiels.



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.