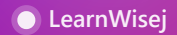


Authentifizierung und Session-Identität

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Login-Abläufe, Umgang mit Passwörtern, ein SessionContext mit verifizierter Identität, Timeouts, Logout, „Angemeldet bleiben“ und Session Fixation. Was das für Entwickler bedeutet, die eine Wisej.NET-App im Produktivbetrieb absichern – und wie Sie dieses Modul lesen.



Fokus: Produktivbetrieb

Login-Abläufe, Umgang mit Passwörtern, ein SessionContext mit verifizierter Identität, Timeouts, Logout, „Angemeldet bleiben“ und Session Fixation.

1. Das Problem im Produktivbetrieb

AccessOps enthält Mitarbeiterdatensätze, vertrauliche Dokumente und Genehmigungsanträge – der erste Bildschirm, den jemand erreicht, ist also auch die erste Kontrolle, die zählt. Alles, was der Rest dieses Kurses durchsetzt – Berechtigungen, Regeln auf Datensatzebene, Audit-Trails –, beruht auf einer einzigen Annahme: dass der Server weiß, wer am anderen Ende dieser Session sitzt, und das belegen kann. Modul 1 hat die Vertrauensgrenzen gezogen; dieses Modul legt eine verifizierte Identität innerhalb der Session-Grenze ab und verhindert, dass sie erraten, ausgeliehen oder über ihre Lebensdauer hinaus genutzt wird.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
gesalzener, langsamer Hash	Eine Einwegableitung des Passworts mit einem zufälligen Salt pro Benutzer und einer bewusst aufwendigen Iterationszahl, sodass eine gestohlene Tabelle nicht umgekehrt werden kann und jeder Rateversuch echte Rechenarbeit kostet.
Vergleich in konstanter Zeit	Zwei Hashes werden mit <code>CryptographicOperations.FixedTimeEquals</code> in einer festen Anzahl von Operationen verglichen, sodass die Antwortzeit des Servers nie verrät, wie viele Bytes eines Rateversuchs stimmten.
SessionContext	Das Objekt, das erst nach der Verifizierung eines Passworts erzeugt wird; es enthält Mitarbeiter-ID, Abteilung, Rollen und Anmeldedaten und liegt in <code>Application.Session</code> als der eine Ort, an dem jeder Code nachfragt, wer der Benutzer ist.
Session-Timeout	Die Leerlaufzeit, nach der Wisej.NET die serverseitige Session samt allem, was sie enthält, entsorgt – für die Anwendung in <code>Default.json</code> konfiguriert, statt pro Bildschirm geschätzt.
Session Fixation	Ein Angriff, bei dem der Bezeichner, der die anonyme Login-Seite getragen hat, anschließend auch die authentifizierte Session trägt, sodass ein vom Angreifer untergeschobener Bezeichner zu einem angemeldeten wird.
Login-Drosselung	Fehlversuche werden pro Konto und pro Client-Adresse gezählt und weitere Versuche für eine Sperrfrist abgewiesen – so wird aus einer unbegrenzten Ratemaschine eine Handvoll Versuche pro Stunde.
Login-Audit-Eintrag	Eine Zeile pro Versuch mit dem versuchten Benutzernamen, der Uhrzeit, der Client-Adresse und dem tatsächlichen Ergebnis – der Ort, an dem der konkrete Fehlergrund festgehalten wird, da der Bildschirm immer nur einen generischen anzeigt.

3. Der rote Faden der Lektion

Ein Passwort prüfen, das Sie nie speichern

AccessOps speichert keine Passwörter. Für jeden Mitarbeiter speichert es drei Werte: den verwendeten Algorithmus samt Iterationszahl, ein zufälliges Salt aus einem kryptografischen Generator und die abgeleiteten Bytes. Bei der Anmeldung läuft dieselbe Ableitung über das gerade eingetippte Passwort, mit Salt und Iterationszahl dieses Mitarbeiters, und das Ergebnis wird mit den gespeicherten Bytes verglichen. Nichts wird entschlüsselt, weil nichts verschlüsselt wurde – ein Hash funktioniert nur in eine Richtung, und das Salt sorgt dafür, dass zwei Mitarbeiter mit demselben Passwort völlig unterschiedliche gespeicherte Werte erzeugen.

Zwei Implementierungen kommen hier infrage. PBKDF2 über `Rfc2898DeriveBytes` bietet explizite Kontrolle: SHA-256, ein 16-Byte-Salt, eine 32-Byte-Ausgabe und eine so hohe Iterationszahl, dass eine Verifizierung einen spürbaren Bruchteil einer Sekunde dauert. Der `PasswordHasher` von ASP.NET Core Identity packt dagegen Algorithmuskennung, Iterationszahl, Salt und Hash in einen String und liefert ein `PasswordVerificationResult`, dessen Wert `SuccessRehashNeeded` Ihnen sagt, dass Sie mit den heutigen

Parametern neu ableiten sollen. So oder so ist der Aufwand der eigentliche Zweck: Jeder Rateversuch eines Angreifers muss eine vollständige Ableitung bezahlen.

Das Passwort selbst lebt genau einen Methodenaufruf lang. Es wird aus der Login- `TextBox` gelesen, an `PasswordService.Verify` übergeben und vergessen: nie ein Feld am Fenster, nie ein Wert in der Audit-Zeile, nie Teil einer Exception-Meldung und nie eine Logzeile – nicht einmal gekürzt.

```
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);
}
```

Eine verifizierte Identität, und nur eine

Wenn `Verify` true zurückgibt, baut der Login-Service einen `SessionContext` : die Mitarbeiter-ID, den Anzeigenamen, die Abteilungs-ID, ob diese Person Mitarbeitende führt, die in diesem Moment aus der Datenbank ermittelten Rollen, die Anmeldezeit und die Client-Adresse. Er wird einmal in `Application.Session` abgelegt, das pro Browser-Session existiert, sodass nichts zwischen der Betriebsleiterin und dem Sachbearbeiter der Lohnbuchhaltung hin- und hergeht. Ein statisches Feld würde von allen Sessions im Prozess geteilt.

Von da an ist er die einzige Quelle der Wahrheit. Bildschirme halten keine Kopie der Benutzer-ID, und Services nehmen keine Aufrufer-ID als Parameter entgegen, denn einen Parameter wählt der aufrufende Code – `DocumentService` liest den Kontext selbst. Wenn das Dokumenten-Grid eine Abteilung anzeigt, stammt diese Abteilung aus dem Kontext und nicht aus einer Combobox, die der Browser hätte ändern können. Das ist die Grundhaltung des Kurses in Code gegossen: Eine Regel, die nur in dem existiert, was die UI anbietet, ist keine Regel.

Der Kontext wird nach der Verifizierung erzeugt und nie davor. Es gibt keinen anonymen Kontext mit leerer ID, der darauf wartet, befüllt zu werden, denn einem Objekt, das in beiden Zuständen existiert, wird irgendwann im falschen Zustand vertraut. Jeder AccessOps-Bildschirm außer dem Login verlangt den Kontext und schickt den Browser zurück, wenn er fehlt; ein typisierter Accessor, der eine Exception wirft, wenn die Session nichts enthält, macht daraus eine Gewohnheit statt einer Review-Anmerkung.

Timeout, Leerlauf-Logout und ein Logout, der es ernst meint

Eine Wisej.NET-Session ist Serverzustand, und ihre Lebensdauer ist Konfiguration, nicht Code. Das Session-Timeout wird in der `Default.json` der Anwendung festgelegt, und wenn es abläuft, entsorgt der Server die Session zusammen mit den Fenstern, den Controls und dem `SessionContext` darin. Wählen Sie einen Wert, den das Unternehmen vertreten kann – zwanzig Minuten passen zu einem Portal, das Gehaltsbänder und unterschriebene Genehmigungen zeigt –, und halten Sie ihn kürzer als jedes Proxy-Timeout vor der App, damit tatsächlich das Verhalten greift, das Sie konfiguriert haben.

Der Leerlauf-Logout ist die sichtbare Hälfte derselben Regel. Ein `Wisej.Web.Timer`, der bei Benutzeraktivität neu startet, warnt eine Minute vor Ablauf und führt dann dieselbe Aufräumroutine aus wie der Logout-Button: Kontext leeren, alles über die Person Zwischengespeicherte verwerfen, den Audit-Eintrag schreiben und `Application.Exit()` aufrufen. Beide Wege rufen eine einzige Methode auf, denn zwei Aufräumroutinen driften auseinander, und diejenige, die niemand anklickt, ist die, die etwas vergisst.

- Die Session endet auf dem Server – den Browser-Tab zu schließen ist kein Logout.
- Jeder Bildschirmzustand, der eine Person nannte, ist verschwunden, bevor die Session entsorgt wird.
- Der Browser landet auf der Login-Seite, und der Schritt zurück im Verlauf zeigt keine Daten.

Brute Force, Credential Stuffing und Fixation

Eine ungedrosselte Login-Seite ist eine unbegrenzte Ratemaschine. `LoginThrottle` führt Zähler pro Konto und pro Client-Adresse; nach fünf Fehlversuchen innerhalb eines kurzen Zeitfensters wird das Konto für eine Sperrfrist abgewiesen, und bei Erfolg werden die Zähler zurückgesetzt. Geben Sie außerdem jeder Fehlerantwort nicht nur denselben Wortlaut, sondern auch dieselbe Form: Wenn der Benutzername nicht existiert, prüfen Sie trotzdem gegen feste Dummy-Zugangsdaten, damit ein fehlendes Konto und ein falsches Passwort gleich viel Zeit kosten.

Die Meldung auf dem Bildschirm ist für jeden Fehlschlag ein und derselbe generische Satz – unbekannter Benutzer, falsches Passwort, gesperrt, deaktiviert. Der wahre Grund gehört in die Audit-Zeile, wo ein Administrator dreißig Versuche mit unbekannten Benutzern von einer Adresse in zehn Minuten sehen kann. Dieses Muster ist Credential Stuffing: Zugangsdatenpaare aus dem Datenleck eines anderen Anbieters, massenhaft ausprobiert und gestoppt durch den Zähler pro Adresse statt durch den pro Konto, da jedes Konto womöglich nur ein einziges Mal probiert wird.

Session Fixation ist die subtilere Bedrohung. Der Bezeichner, der die anonyme Login-Seite getragen hat, darf nicht die verifizierte Identität tragen; die Session vor dem Login endet daher nach der Verifizierung, und der Browser arbeitet in einer neuen weiter, ohne dass etwas übernommen wird. „Angemeldet bleiben“ folgt derselben Disziplin: nie ein gespeichertes Passwort und nie ein erratbarer Wert, sondern ein langes, zufälliges Token pro Gerät, im Ruhezustand gehasht, an einen Mitarbeiter gebunden, mit Ablaufdatum versehen, über die Admin-Konsole widerrufbar und vor dem Neuaufbau des SessionContext erneut gegen das aktive Konto geprüft.



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:

- ✓ eine Login-Seite implementieren, die Zugangsdaten gegen einen gesalzenen Hash prüft und das Passwort nie speichert.

- ✓ einen SessionContext erstellen, der eine verifizierte Identität in Application.Session hält und die einzige Quelle der Wahrheit dafür ist, wer der Benutzer ist.
- ✓ Session-Timeout, Leerlauf-Logout und expliziten Logout so konfigurieren, dass der Zustand entsorgt wird und nicht wiederverwendet werden kann.
- ✓ den Login-Ablauf vor Brute Force, Credential Stuffing und Session Fixation schützen.

Lab- und Prüfungsleitfaden

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



Praxis-Lab

Implementieren Sie die Authentifizierung von AccessOps: eine Login-Seite mit Sperre nach fünf Fehlversuchen, Passwortprüfung mit einem modernen gesalzenen Hash, einen SessionContext, der erst nach der Verifizierung erzeugt und in der Session abgelegt wird, ein Leerlauf-Timeout, das den Benutzer zum Login zurückschickt und den Zustand leert, einen expliziten Logout, der die Session beendet, und einen Audit-Eintrag für jeden Anmeldeversuch.

Ausgangspunkt

Sie öffnen die AccessOps-Solution, die Sie in Modul 1 angelegt haben und die noch ein Grundgerüst ist: ein Startfenster, eine Platzhalter-Login-Seite, deren Button einfach die Hauptkonsole öffnet, und das Bedrohungsmodell, das die Authentifizierung als erste zu bauende Kontrolle benannt hat. Die Mitarbeitertabelle existiert mit Namen und Abteilungen, aber ohne Zugangsdaten, und nichts in der Solution weiß bisher, wer angemeldet ist.

Vorgeschlagenes Vorgehen

- ✓ Schreiben Sie zuerst den `PasswordService` mit einer Hash-Methode, die ein frisches zufälliges Salt erzeugt, und einer Verify-Methode, die mit dem gespeicherten Salt und der gespeicherten Iterationszahl neu ableitet und mit einem Gleichheitsvergleich in konstanter Zeit vergleicht; ergänzen Sie die Mitarbeitertabelle um Spalten für Salt, Iterationszahl und Hash und legen Sie über den Service zwei Testkonten an.
- ✓ Definieren Sie `SessionContext` mit Mitarbeiter-ID, Anzeigename, Abteilungs-ID, Manager-Kennzeichen, Rollen und Anmeldezeit, dazu einen typisierten Accessor über `Application.Session`, der den aktuellen Kontext liefert und eine Exception wirft, wenn niemand angemeldet ist.
- ✓ Bauen Sie `LoginService.SignIn` als einzigen Weg, auf dem Zugangsdaten geprüft werden: `LoginThrottle` fragen, ob das Konto gesperrt ist, das Passwort prüfen, für jedes Ergebnis den Audit-Eintrag schreiben und bei Erfolg die Zähler zurücksetzen, die Session vor dem Login beenden und den SessionContext für die neue Session erzeugen.
- ✓ Bauen Sie die Login-Seite so um, dass ihr Button-Handler nur `LoginService` aufruft und auf das Ergebnis reagiert – ein generischer Satz für jeden Fehlschlag, die Konsole bei Erfolg –, und sorgen Sie dafür, dass jeder andere Bildschirm den Kontext verlangt und zum Login umleitet, wenn er fehlt.
- ✓ Legen Sie das Session-Timeout in `Default.json` fest, ergänzen Sie den Leerlauf-Timer mit seiner Warnung eine Minute vorher und schreiben Sie eine einzige Logout-Routine, die den Kontext leert, die Abmeldung protokolliert und `Application.Exit()` aufruft – aufgerufen sowohl vom Button als auch vom Leerlaufpfad.

Geforderte Ergebnisse

- ✓ Login-Seite mit Sperre und generischen Fehlermeldungen
- ✓ Service für Passwort-Hashing und -Verifizierung
- ✓ SessionContext mit verifizierter Identität
- ✓ Leerlauf-Timeout und expliziter Logout
- ✓ Audit-Einträge für Anmeldungen

Wiederholungsfragen

- ✓ Ihre Login-Seite beantwortet ein falsches Passwort in 480 ms und einen unbekannten Benutzernamen in 6 ms. Erklären Sie, was ein Angreifer allein aus diesem Unterschied erfährt, und beschreiben Sie die zwei Änderungen, die das Signal beseitigen, ohne die Meldung zu ändern.
- ✓ Ein Kollege schlägt vor, den angemeldeten Mitarbeiter in einer statischen Eigenschaft zu halten, weil „das einfacher ist, als jedes Mal in die Session zu greifen“. Beschreiben Sie genau, was der zweite gleichzeitige Benutzer sieht und in welchem Moment sich die beiden Sessions zum ersten Mal in die Quere kommen.
- ✓ Ihr Logout-Button leert den SessionContext, beendet die Session aber nicht. Nennen Sie zwei Dinge, die danach noch wiederherstellbar sind, und sagen Sie, welches Sie in einem Portal mit vertraulichen Dokumenten für das gravierendere halten.



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.