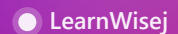


Autorisation : rôles, claims et permissions

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



Guide de la leçon

Contrôles fondés sur les rôles et sur les claims, services de permissions, application des règles dans les services et les repositories, et adaptation de l'UI à ce que l'utilisateur a le droit de faire. Ce que cela implique pour un développeur qui sécurise une application Wisej.NET de production — et comment aborder ce module.



Enjeux en production

Contrôles fondés sur les rôles et sur les claims, services de permissions, application des règles dans les services et les repositories, et adaptation de l'UI à ce que l'utilisateur a le droit de faire.

1. Le problème en production

AccessOps sait désormais qui est connecté : le Module 2 a laissé une identité vérifiée dans le `SessionContext`. Savoir qui est quelqu'un ne dit rien de ce qu'il peut faire d'un document confidentiel ou d'une demande d'approbation à six chiffres. Dans un portail interne, l'incident type est rarement une connexion défaillante — c'est un gestionnaire de paie qui peut lire les contrats du service juridique parce que la requête n'a jamais été restreinte, ou un chef d'équipe qui approuve sa propre note de frais parce que le seul contrôle était un bouton désactivé.

2. Concepts clés

Concept	Pourquoi c'est important
rôle	Une fonction métier nommée, à gros grain, comme Employee, Manager ou Auditor. Les rôles sont pratiques à administrer mais trop grossiers pour être testés directement, car un écran a besoin de connaître une action, pas un métier entier.
claim	Un fait nom/valeur sur l'identité vérifiée — département 14, centre de coût EU-North, manager de l'équipe 7. Les claims portent les données dont les règles au niveau des enregistrements ont besoin, et ils proviennent du référentiel d'identités, jamais du navigateur.
permission	Une action nommée que l'application peut effectuer, comme <code>ApproveRequest</code> ou <code>ViewDepartmentDocuments</code> . Le code teste des permissions ; les administrateurs attribuent des rôles ; la correspondance entre les deux relève de la configuration.
résolution des permissions	L'étape unique, à la connexion, qui développe les rôles et les claims en un ensemble de permissions plat et immuable conservé dans le <code>SessionContext</code> , afin qu'aucune requête ne relise la table des rôles et que deux contrôles ne puissent jamais se contredire.
garde d'autorisation	Un appel tel que <code>Require(Permission.ApproveRequest)</code> en tête d'une méthode de service, qui lève une exception avant que le moindre travail ne commence. C'est le seul endroit où l'application de la règle a réellement lieu.
règle au niveau des enregistrements	Une restriction sur les lignes qu'une action autorisée peut toucher — votre propre département, vos propres collaborateurs directs — appliquée dans la requête ou la commande plutôt qu'après le chargement des données.
adaptation de l'UI	Masquer, désactiver ou expliquer une action à partir du même ensemble de permissions que celui appliqué par le serveur. C'est une attention envers l'utilisateur, jamais un contrôle.

3. Déroulé de la leçon

Des rôles et des claims en entrée, des permissions en sortie

Commencez par séparer les trois vocabulaires. Les administrateurs d'AccessOps raisonnent en rôles : un employé est Employee, Manager, DepartmentAdmin ou Auditor. Le référentiel d'identités porte aussi des claims — le département de l'employé, le centre de coût, la liste des collaborateurs directs. Le code de l'application, en revanche, ne devrait jamais demander `if (role == "Manager")`, car le jour où les auditeurs financiers seront autorisés à approuver, vous modifierez quarante écrans au lieu d'une table.

Définissez plutôt une énumération `Permission` d'actions : `ViewDepartmentDocuments`, `ViewAllDocuments`, `UploadDocument`, `SubmitRequest`, `ApproveRequest`, `ViewAuditTrail`, `ManageUsers`. Un `PermissionService` détient la correspondance entre rôles et permissions, applique les octrois et les refus propres à chaque employé issus des claims, et produit un ensemble plat unique. Les refus l'emportent sur les octrois, et le résultat est trié et immuable, pour que rien en aval ne puisse le modifier.

Résolvez-le une seule fois, juste après la vérification des identifiants, et stockez-le sur le `SessionContext` qui vit déjà dans `Application.Session`. Ce moment compte : une session Wisej.NET est un objet serveur de longue durée, si bien qu'une résolution à chaque clic surchargerait la base de données, tandis qu'une résolution paresseuse ouvre la porte à deux réponses différentes au sein d'un même processus métier. La contrepartie, c'est l'obsolescence — une permission révoquée ne prend effet qu'à la connexion suivante — c'est pourquoi AccessOps enregistre un horodatage `PermissionsResolvedAt` et dote la console d'administration d'une action « force re-resolve » qui reconstruit l'ensemble pour une session donnée.

La garde en tête de chaque méthode

L'application des règles se fait à un endroit unique et prévisible : la première ligne exécutable de chaque méthode publique de service. `DocumentService` et `ApprovalService` reçoivent tous deux une garde `IAuthorize` et le `SessionContext` ; chaque méthode commence par `Require`, qui lève une `NotAuthorizedException` lorsque la permission manque. Comme la garde lève une exception au lieu de renvoyer un indicateur, un résultat qu'on aurait oublié de tester ne peut pas laisser passer silencieusement une commande.

Placez la garde dans le service, pas dans le gestionnaire du bouton, et pas uniquement dans le repository. Le gestionnaire n'est qu'un appelant parmi d'autres — une tâche en arrière-plan lancée avec `Application.StartTask`, un rappel planifié, un handler HTTP personnalisé et le callback OpenID Connect du module suivant atteignent tous les mêmes méthodes, et un contrôle écrit dans un gestionnaire `Click` n'en protège qu'un seul. Les repositories sont le mauvais endroit pour la raison inverse : ils doivent pouvoir être réutilisés par des appelants aux droits différents, et un repository qui consulte la session est impossible à tester.

La garde doit aussi rester rigoureuse sur l'identité. Elle lit l'utilisateur dans le `SessionContext`, jamais dans un paramètre, pour qu'un appelant ne puisse pas transmettre l'id d'un autre employé et hériter de ses droits. Chaque refus est journalisé avec l'utilisateur, la permission et l'enregistrement ciblé avant que l'exception ne se propage, car une rafale de refus est exactement ce à quoi ressemble une attaque vue du serveur.

Des règles qui vivent dans la requête

Une permission répond à la question « cette personne peut-elle approuver des demandes, tout court » ; elle ne dit rien de *lesquelles*. AccessOps a besoin de deux règles au niveau des enregistrements en plus : un employé ne voit que les documents de son propre département, et un manager ne peut approuver que les demandes émises par ses collaborateurs directs. Ce sont les règles qu'un feature flag ne peut pas exprimer, et celles que l'on greffe le plus souvent après coup, une fois les données déjà chargées.

Appliquez-les dans la requête. Le repository des documents prend le département dans le contexte et l'intègre à la clause `Where`, de sorte que les lignes restreintes ne quittent jamais la base de données, n'apparaissent jamais dans un comptage et n'atteignent jamais une grille susceptible de les paginer. L'alternative — tout charger puis filtrer le `DataSource` ensuite — a déjà payé le prix de l'exposition des données à un processus qui peut les journaliser, et c'est précisément ce qu'une modification ultérieure de la pagination a tendance à défaire.

Les commandes sont restreintes de la même manière. L'approbation récupère la demande via une méthode du repository qui exige à la fois l'id de la demande et l'id d'employé de l'approbateur, et qui

renvoie null lorsque la demande n'entre pas dans le périmètre de ce manager : un identifiant hors périmètre est ainsi impossible à distinguer d'un identifiant inexistant.

```
public async Task ApproveAsync(int requestId, string note)
{
    _authorize.Require(Permission.ApproveRequest);

    var request = await _requests.GetForApproverAsync(
        requestId, _session.EmployeeId);
    if (request == null)
        throw new UnauthorizedException("Request is outside your scope.");

    request.Approve(_session.EmployeeId, note, DateTime.UtcNow);
    await _requests.SaveAsync(request);
}
```

Une UI qui dit la vérité sans qu'on lui fasse confiance

Une fois le serveur solide, l'interface peut enfin se rendre utile. AccessOps lit le même ensemble de permissions lorsqu'il construit chaque écran : l'entrée de la console d'administration est retirée de la navigation en l'absence de `ManageUsers`, le bouton d'upload est masqué sans `UploadDocument`, et le bouton Approve d'une demande sur laquelle l'utilisateur ne peut pas agir est désactivé avec une courte explication — « only the requester's manager can approve » — au lieu de ne rien faire en silence.

- Masquez ce que l'utilisateur ne pourra jamais faire dans cette application.
- Désactivez et expliquez ce qu'il pourrait faire sur un autre enregistrement, ou avec un autre rôle.
- Ne retirez jamais un contrôle à la place d'écrire la vérification côté serveur.

La distinction entre masquer et désactiver est une décision d'ergonomie, pas de sécurité. Dans une application Wisej.NET, les contrôles sont des objets C# côté serveur : un `Button` masqué n'est donc réellement pas rendu — mais les événements qu'il aurait déclenchés ne sont pas la seule voie d'accès au service, et une session restée ouverte pendant un changement de permissions peut afficher un menu périmé. C'est pourquoi le dernier livrable du lab est un test négatif qui contourne entièrement l'UI : il construit une session pour un non-manager, appelle directement `ApprovalService.ApproveAsync` avec un id de demande falsifié, et vérifie à la fois qu'une `UnauthorizedException` a été levée et que le statut de la demande en base n'a jamais changé.



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 :

- ✓ Modéliser des rôles, des claims et des permissions fines, et les résoudre une seule fois dans le SessionContext.
- ✓ Appliquer l'autorisation dans les services et les repositories, pour que chaque commande et chaque requête vérifie l'appelant.
- ✓ Appliquer des règles au niveau des enregistrements, comme l'appartenance à un département et le périmètre d'un manager, et pas seulement des feature flags.
- ✓ Masquer, désactiver ou expliquer des actions de l'UI en fonction des permissions, sans compter sur l'UI pour faire respecter les règles.

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 l'autorisation à AccessOps : un `PermissionService` qui résout les rôles et les claims en permissions à la connexion, une garde `Authorize` appelée par chaque méthode de `DocumentService` et d'`ApprovalService`, une règle au niveau des enregistrements selon laquelle les employés ne voient que leur département et les managers que leurs collaborateurs directs, une UI qui masque les actions d'administration et désactive les boutons d'approbation que l'utilisateur ne peut pas utiliser, et un test prouvant qu'une requête falsifiée émise par un non-manager est rejetée par le service.

Point de départ

Vous ouvrez la solution AccessOps telle que le Module 2 l'a laissée : une page de connexion qui vérifie un hachage salé, un `SessionContext` qui conserve l'employé vérifié dans `Application.Session`, l'expiration pour inactivité et la déconnexion, et les enregistrements d'audit des connexions. La liste des documents et la file des approbations existent sous forme d'écrans qui affichent actuellement toutes les lignes à tout le monde, et la table des rôles attribue aux employés les rôles `Employee`, `Manager`, `DepartmentAdmin` et `Auditor` sans que rien ne la lise encore.

Approche suggérée

- ✓ Définissez une énumération `Permission` pour les actions qu'AccessOps effectue réellement, et une correspondance rôles-permissions placée dans la configuration plutôt que dans le code, pour qu'un nouveau rôle soit une simple modification de données.
- ✓ Écrivez `PermissionService.ResolveFor(employee)` pour développer les rôles, appliquer les octrois et refus propres à l'employé issus des claims, le refus l'emportant, et renvoyer un ensemble immuable que le flux de connexion rattache au `SessionContext` avec le département et la liste des collaborateurs directs.
- ✓ Ajoutez une garde `Authorize` qui lit le `SessionContext` courant et lève `NotAuthorizedException`, puis appelez `Require` en première ligne de chaque méthode publique de `DocumentService` et d'`ApprovalService`.
- ✓ Descendez les règles au niveau des enregistrements dans les repositories : intégrez le filtre de département à la requête des documents, et dotez le repository des approbations d'une recherche qui prend l'id de la demande et l'id d'employé de l'approbateur et renvoie null lorsque la demande n'émane pas des collaborateurs directs de ce manager.
- ✓ Liez les écrans à l'ensemble de permissions — retirez la console d'administration de la navigation, masquez l'upload, désactivez Approve avec une raison sur les lignes où l'utilisateur ne peut pas agir — puis écrivez le test négatif qui appelle directement le service avec un id de demande falsifié et vérifie que rien n'a changé.

Livrables attendus

- ✓ `PermissionService` résolvant les permissions dans le `SessionContext`

- ✓ Garde d'autorisation dans chaque méthode de service
- ✓ Règles au niveau des enregistrements par département et par manager
- ✓ Adaptation de l'UI selon les permissions
- ✓ Test négatif avec une requête falsifiée

Questions de révision

- ✓ Un auditeur reçoit des droits d'approbation alors que sa session AccessOps est encore ouverte. Expliquez précisément à quel moment la nouvelle permission prend effet avec votre conception, pourquoi vous avez choisi ce moment, et ce qu'il faudrait modifier pour qu'elle soit immédiate.
- ✓ Votre requête de documents filtre par département dans le repository. Décrivez ce qu'un attaquant gagne si ce filtre est déplacé vers le `DataSource` de la grille, alors même que le résultat visible à l'écran est identique.
- ✓ `ApprovalService` comme son repository pourraient vérifier que l'approbateur est le manager du demandeur. Défendez l'un des deux comme le bon emplacement, et dites ce qui casse si la vérification se trouve dans l'autre, ou dans les deux.



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.