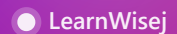


Authorization: Roles, Claims and Permissions

Lesson Guide · Lab / Exam Guide



Lesson Guide

Role-based and claim-based checks, permission services, enforcing rules in services and repositories, and adapting the UI to what the user may do. What this means for a developer securing a production Wisej.NET app — and how to read this module.



Production focus

Role-based and claim-based checks, permission services, enforcing rules in services and repositories, and adapting the UI to what the user may do.

1. The production problem

AccessOps now knows who is signed in: Module 2 left a verified identity in the `SessionContext`. Knowing who someone is answers nothing about what they may do with a confidential document or an approval request worth six figures. The failure mode in an internal portal is rarely a broken login — it is a payroll clerk who can read the legal department's contracts because the query was never scoped, or a team lead who approves their own expense request because the only check was a disabled button.

2. Key concepts

Concept	Why it matters
role	A coarse named job function such as Employee, Manager or Auditor. Roles are convenient to administer but too blunt to test against directly, because a screen needs to know about one action, not a whole job.
claim	A name/value fact about the verified identity — department 14, cost centre EU-North, manager of team 7. Claims carry the data that record-level rules need, and they come from the identity store, never from the browser.
permission	One named action the application can perform, such as <code>ApproveRequest</code> or <code>ViewDepartmentDocuments</code> . Code tests permissions; administrators assign roles; the mapping between them is configuration.
permission resolution	The single step at login that expands roles and claims into a flat, immutable permission set held in the <code>SessionContext</code> , so no request re-reads the role table and no two checks can disagree.
authorization guard	A call such as <code>Require(Permission.ApproveRequest)</code> at the top of a service method that throws before any work happens. It is the only place enforcement actually occurs.
record-level rule	A restriction on which rows a permitted action may touch — your own department, your own reports — applied inside the query or command rather than after the data has already been loaded.
UI adaptation	Hiding, disabling or explaining an action based on the same permission set the server enforces. It is a courtesy to the user and never a control.

3. Lesson narrative

Roles and claims in, permissions out

Start by separating the three vocabularies. AccessOps administrators think in roles: an employee is an Employee, a Manager, a DepartmentAdmin or an Auditor. The identity store also carries claims — the employee's department, the cost centre, the list of direct reports. Application code, however, should never ask `if (role == "Manager")`, because the day finance auditors are allowed to approve, you are editing forty screens instead of one table.

Instead define a `Permission` enumeration of actions: `ViewDepartmentDocuments`, `ViewAllDocuments`, `UploadDocument`, `SubmitRequest`, `ApproveRequest`, `ViewAuditTrail`, `ManageUsers`. A `PermissionService` owns the mapping from roles to permissions, applies per-employee grants and denials from claims, and produces one flat set. Denials win over grants, and the result is sorted and immutable so nothing downstream can mutate it.

Resolve it exactly once, immediately after the credentials are verified, and store it on the `SessionContext` that already lives in `Application.Session`. That timing matters: a Wisej.NET session is a long-lived server object, so resolving per click would hammer the database, while resolving lazily invites two different answers inside one workflow. The trade is staleness — a revoked permission does not take effect until the

next sign-in — so AccessOps records a `PermissionsResolvedAt` timestamp and gives the admin console a "force re-resolve" action that rebuilds the set for a named session.

The guard at the top of every method

Enforcement lives in one predictable place: the first executable line of every public service method.

`DocumentService` and `ApprovalService` both take an `IAuthorize` guard and the `SessionContext`; every method begins with `Require`, which throws a `NotAuthorizedException` when the permission is missing. Because it throws rather than returns a flag, a forgotten result check cannot silently let a command through.

Put the guard in the service, not in the button handler, and not only in the repository. The handler is one of several callers — a background job started with `Application.StartTask`, a scheduled reminder, a custom HTTP handler and next module's OpenID Connect callback all reach the same methods, and a check written in a `Click` handler protects exactly one of them. Repositories are the wrong home for the opposite reason: they should be reusable across callers with different rights, and a repository that consults the session is impossible to test.

Keep the guard honest about identity too. It reads the user from `SessionContext`, never from a parameter, so a caller cannot pass someone else's employee ID and inherit their rights. Every rejection is logged with the user, the permission and the target record before the exception propagates, because a burst of these is what an attack looks like from the server side.

Rules that live inside the query

A permission answers "may this person approve requests at all"; it says nothing about *which* ones.

AccessOps needs two record-level rules on top: an employee sees only documents belonging to their own department, and a manager may approve only requests raised by their direct reports. These are the rules a feature flag cannot express, and the ones most often bolted on after the data has already been loaded.

Apply them inside the query. The document repository takes the department from the context and composes it into the `Where` clause, so the restricted rows never leave the database, never appear in a count and never reach a grid that might page through them. The alternative — loading everything and filtering the `DataSource` afterwards — has already paid the cost of exposing the data to a process that can log it, and it is exactly what a later paging change tends to undo.

Commands scope the same way. Approving fetches the request through a repository method that requires both the request ID and the approver's employee ID and returns null when the request is not in that manager's scope, so an out-of-scope identifier is indistinguishable from a missing one.

```
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);
}
```

A UI that tells the truth without being trusted

With the server sound, the interface can finally be helpful. AccessOps reads the same permission set when it builds each screen: the admin console entry is removed from the navigation when `ManageUsers` is absent, the upload button is hidden without `UploadDocument`, and the Approve button on a request the user may not act on is disabled with a short explanation — "only the requester's manager can approve" — rather than silently doing nothing.

- Hide what the user can never do in this application at all.
- Disable and explain what they could do on another record, or with another role.
- Never remove a control instead of writing the server-side check.

The distinction between hiding and disabling is a usability decision, not a security one. In a Wisej.NET app the controls are server-side C# objects, so a hidden `Button` genuinely is not rendered — but the events it would have raised are not the only way into the service, and a session that has been open across a permission change may be showing a stale menu. That is why the lab's last deliverable is a negative test that bypasses the UI entirely: it builds a session for a non-manager, calls `ApprovalService.ApproveAsync` directly with a forged request ID, and asserts both that `UnauthorizedException` was thrown and that the request's status in the database never changed.



Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.



Learning objectives

By the end of this module you'll be able to:

- ✓ Model roles, claims and fine-grained permissions and resolve them once into the `SessionContext`.
- ✓ Enforce authorization inside services and repositories so every command and query checks the caller.

- ✓ Apply record-level rules such as department ownership and manager scope, not just feature flags.
- ✓ Hide, disable or explain UI actions based on permissions without relying on the UI for enforcement.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Add authorization to AccessOps: a `PermissionService` resolving roles and claims into permissions at login, an `Authorize` guard called by every `DocumentService` and `ApprovalService` method, a record-level rule that employees see only their department and managers only their reports, a UI that hides admin actions and disables approve buttons the user cannot use, and a test proving a forged request from a non-manager is rejected by the service.

Starting point

You open the AccessOps solution as Module 2 left it: a login page that verifies a salted hash, a `SessionContext` holding the verified employee in `Application.Session`, idle timeout and logout, and login audit records. The documents list and the approval queue exist as screens that currently show every row to everyone, and the roles table has employees assigned to Employee, Manager, DepartmentAdmin and Auditor with nothing reading it yet.

Suggested approach

- ✓ Define a `Permission` enumeration for the actions AccessOps really performs, and a role-to-permission map in configuration rather than in code, so a new role is a data change.
- ✓ Write `PermissionService.ResolveFor(employee)` to expand roles, apply per-employee grants and denials from claims with denial winning, and return an immutable set that the login flow attaches to the `SessionContext` together with the department and the list of direct reports.
- ✓ Add an `Authorize` guard that reads the current `SessionContext` and throws `NotAuthorizedException`, then call `Require` as the first line of every public method on `DocumentService` and `ApprovalService`.
- ✓ Push the record-level rules into the repositories: compose the department filter into the document query, and give the approval repository a lookup that takes the request ID and the approver's employee ID and returns null when the request is outside that manager's reports.
- ✓ Bind the screens to the permission set — drop the admin console from the navigation, hide upload, disable Approve with a reason on rows the user may not act on — then write the negative test that calls the service directly with a forged request ID and asserts nothing changed.

Required deliverables

- ✓ `PermissionService` resolving permissions into the `SessionContext`
- ✓ Authorization guard in every service method
- ✓ Department and manager record-level rules
- ✓ Permission-aware UI adaptation
- ✓ Negative test with a forged request

Student review questions

- ✓ An auditor is granted approval rights while their AccessOps session is still open. Explain exactly when the new permission takes effect with your design, why you chose that moment, and what you would have to change to make it immediate.
- ✓ Your document query filters by department in the repository. Describe what an attacker gains if that filter is moved to the grid's `DataSource` instead, even though the visible result on screen is identical.
- ✓ Both `ApprovalService` and its repository could check that the approver manages the requester. Argue for one of them as the right home, and say what breaks if the check lives in the other, or in both.



Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.