TUTORIAL

ASP.NET Core Authentication and Authorization Tutorial

Learn how to implement ASP.NET Core authentication and authorization with a practical workflow: prerequisites, setup, policy-based access control, validation, and production checks.

Programming - August 03, 2026

What you will build

A typical ASP.NET Core application needs two separate security decisions:

- Authentication answers, ?Who is the caller??
- Authorization answers, ?Is this caller allowed to do this action??

This tutorial shows how to wire both into an ASP.NET Core app so that authenticated users can sign in, protected endpoints reject anonymous requests, and authorization policies control access based on roles or claims. By the end, you will know how to implement the flow, verify it locally, and check the pieces that commonly fail before production.

The examples below assume a server-side ASP.NET Core app or API with the built-in middleware pipeline. If your app uses external identity providers, the same authorization model still applies, but the authentication configuration source changes.

Prerequisites and stop-here checks

Before you start, confirm the following:

- You have a working ASP.NET Core project targeting a supported runtime.

- You can edit Program.cs and app settings.
- You know whether your app will use cookies, bearer tokens, or an external identity provider.
- You understand which endpoints must be public and which must be protected.

Stop here if: you do not yet know how users will authenticate. Authorization rules depend on the shape of the authenticated principal, so you need the authentication scheme decided first. If you skip that decision, you may build policies that look correct but never receive the claims or roles they expect.

Stop here if: you do not know which identity source is authoritative for roles and claims. Mixing local role definitions with external claims without a clear mapping is a common cause of authorization failures.

Step 1: Add the authentication and authorization pipeline

Goal

Register the services and middleware that make authentication and authorization available to the app.

Action

In Program.cs, add authentication and authorization services, then place the middleware in the correct order:

If your application uses JWT bearer tokens instead of cookies, the registration changes, but the placement rule does not: authentication must run before authorization.

Expected output

The app can create an authenticated user principal and evaluate authorization rules for incoming requests.

Validation

Check that:

- UseAuthentication() appears before UseAuthorization().
- Authentication is configured with the scheme your app actually uses.
- Authorization policies are registered before the app starts handling requests.

A quick sanity test is to hit a protected endpoint without credentials and confirm you receive a challenge or denial, not a successful response.

Common failure

The most common failure is middleware order. If authorization runs before authentication, the user principal will not be populated in time, and policies may fail unexpectedly or behave as if every request is anonymous.

Step 2: Protect endpoints with authorization attributes

Goal

Prevent anonymous access to sensitive routes and make the protection visible in code.

Action

Apply [Authorize] to controllers, actions, Razor pages, or minimal API endpoints that require authentication:

Allow public access explicitly with [AllowAnonymous] only where needed, such as login pages or health checks.

Expected output

Protected endpoints reject anonymous requests, while public endpoints remain accessible.

Validation

Verify that:

- A request without credentials is blocked.
- A valid authenticated request reaches the action.
- Only intended endpoints are marked anonymous.

For APIs, test with and without the token or cookie that represents the authenticated user.

Common failure

A frequent mistake is assuming a route is protected because the application ?uses authentication.? Authentication alone only identifies the user; it does not block access unless authorization is applied.

Step 3: Use policies instead of scattered role checks

Goal

Centralize access rules so that authorization remains maintainable as the app grows.

Action

Define policies once and reference them where needed. Policies are especially useful when access depends on claims, role combinations, or custom requirements. For more complex patterns, see ASP.NET Core Authorization Policies with Role and Claim Checks.

Then apply the policy to an endpoint:

Expected output

The access rule is defined once and reused consistently across controllers or endpoints.

Validation

Confirm that:

- The policy name matches exactly at registration and use sites.
- The authenticated principal contains the role or claim the policy requires.
- Unauthorized users are denied even when authenticated.

Common failure

Policy typos are easy to miss because the app may still start. The result is often a route that appears secured but never evaluates the intended rule because the wrong policy name was applied.

Step 4: Map claims and roles correctly

Goal

Ensure the authenticated identity carries the claims and roles that your authorization rules expect.

Action

Inspect the principal created by your authentication scheme and verify that role claims and custom claims are present. In many systems, the failure is not in ASP.NET Core itself but in identity mapping: the token or cookie does not contain the expected claim type, or the app is checking the wrong one.

Use a simple diagnostic endpoint during development to inspect the current user:

Expected output

You can confirm the authenticated user has the claims and roles needed by your policies.

Validation

Check that:

- Role claims are issued in the format your app expects.
- Custom claims use stable names and values.
- The principal has a non-empty identity and the expected authentication type.

Common failure

A common mismatch is using a claim in the token but checking a different claim type in the app. Another is storing roles in a custom claim without configuring role resolution correctly.

Step 5: Validate authorization behavior with realistic test cases

Goal

Prove that the protection works for anonymous, authenticated, and unauthorized users.

Action

Run the following test cases against each protected endpoint:

- Anonymous request.
- Authenticated request with the correct role or claim.
- Authenticated request with the wrong role or claim.
- Authenticated request with expired or invalid credentials.

If your application uses request payload validation as well, keep input validation separate from access control. Authorization decides whether the caller may proceed; model validation decides whether the request body is well-formed. For request validation patterns, see ASP.NET Core Request Validation with Data Annotations.

Expected output

Each case produces the response you expect:

- Anonymous requests are challenged or denied.

- Valid authorized requests succeed.
- Authenticated but unauthorized requests fail with an access-denied response.
- Invalid credentials are rejected before access is granted.

Validation

Use a client such as curl, Postman, or an automated test suite to verify each case against the exact endpoint path, not just against a general route pattern.

Common failure

Teams often test only the ?happy path.? That confirms the endpoint works for one allowed user but does not prove that unauthorized users are blocked.

Step 6: Keep authentication and authorization concerns separate

Goal

Avoid design mistakes that make security rules harder to reason about or maintain.

Action

Treat these as separate layers:

- Authentication establishes identity.
- Authorization evaluates permissions.

- Validation checks request shape and data quality.

This separation helps you reason about failures. For example, a malformed request should usually fail validation, not authorization. A user with a valid token but insufficient permissions should fail authorization, not authentication.

If your app needs fine-grained access logic, use policy-based authorization rather than hard-coded role checks inside controllers. If your authorization rules also depend on rate control or abuse prevention, combine them with operational protections such as ASP.NET Core Rate Limiting Middleware for API Protection rather than overloading authorization with traffic management logic.

Expected output

Security behavior becomes predictable and easier to debug.

Validation

When a request fails, identify which layer rejected it:

- Did authentication fail to identify the user?
- Did authorization deny access?
- Did request validation reject the payload?

If those answers are not obvious in logs or traces, add structured logging around the relevant middleware and endpoint filters.

Common failure

A common architectural mistake is using authorization as a catch-all for every invalid request. That makes troubleshooting harder and can hide input problems that should be handled earlier.

Step 7: Review production-readiness checks

Goal

Verify that the app is safe to deploy with the implemented security flow.

Action

Before production, review the following operational items:

- The default authentication scheme is correct for the deployment model.
- Protected routes are explicitly protected.
- Anonymous routes are intentionally anonymous.
- Policies match the claims or roles issued by the identity source.
- Failed access attempts are logged without exposing secrets.
- Redirect behavior is appropriate for browser-based apps and APIs.
- Token expiration, clock skew, and sign-in session lifetime are understood and tested.

If you use an external identity provider, verify the tenant, issuer, audience, and callback configuration in the target environment. Those values often differ between development and production even when the code is identical.

Expected output

You have a predictable security posture, and the application behaves consistently in development, staging, and production.

Validation

Run a short pre-deployment check:

- Sign in with a known valid user.
- Access a protected endpoint.
- Access a denied endpoint.
- Repeat with an anonymous request.
- Repeat with a user lacking the required role or claim.

Confirm that logs show the expected success and failure patterns without leaking tokens, cookies, or personally sensitive identity data.

Common failure

The most damaging mistakes usually come from configuration drift: the app is correct in code but points at the wrong identity settings in production, or the policy expects claims that are not issued in that environment.

Finished state

When this tutorial is implemented correctly, your ASP.NET Core app will:

- authenticate users through the chosen scheme,
- block anonymous access to protected endpoints,
- authorize access with policies that reflect real roles or claims,
- separate access control from request validation,
- and pass a practical pre-production verification workflow.

That is the right operational end state: authentication identifies the caller, authorization enforces access rules, and your validation checks prove the system behaves as intended before it reaches production.

Use this guidance together with Kafka Streams security and Apache Spark data encryption to connect the workflow with related operational context already available on the site.

> Part of the Programming: ASP.NET Insights content cluster.