ARTICLE

Implementing ASP.NET Core JWT Authentication with Refresh Tokens

JWT access tokens are efficient for API authentication, but short lifetimes create a usability problem. Refresh tokens solve that gap when you need secure session renewal without forcing frequent logins. This article explains how the pattern works, where it fits, and what to validate before production use.

Programming - July 09, 2026

Key takeaways

JWT access tokens and refresh tokens solve different problems. The access token is meant for short-lived API authorization, while the refresh token is a longer-lived credential used only to obtain a new access token when the current one expires.

This pattern is useful when you want stateless API requests, predictable token expiry, and fewer interactive logins without extending the lifetime of the bearer token that reaches your API. It is not a universal replacement for cookie sessions, and it becomes risky if refresh tokens are stored, rotated, or revoked poorly.

The operational question is simple: should your ASP.NET Core API use refresh tokens, and if so, what does a safe implementation look like? By the end of this article, you should be able to decide whether the pattern fits your environment, understand the token flow, validate the core security controls, and check the main production readiness risks before rollout.

Why this pattern matters

Short-lived JWT access tokens are common because they reduce exposure if a token is stolen and keep API authorization fast. The trade-off is obvious: when the token expires, the client must either authenticate again or obtain a replacement from a trusted renewal path.

That renewal path is the refresh token. In practice, it is a long-lived credential issued by the identity layer and stored separately from the access token. The client presents it only to a token endpoint, not to the API itself. If the refresh token is valid, the server issues a new access token and usually a new refresh token as well.

This matters operationally because it lets you keep access tokens short without degrading user experience. It also gives you a revocation point. If a device is compromised, a suspicious session can be invalidated by revoking refresh tokens even if previously issued access tokens remain valid until expiry.

For API security design context, it is worth pairing this topic with Securing ASP.NET Core APIs with JWT Authentication and Claims Validation, especially if you are defining issuer, audience and claim validation rules around the same authentication stack.

How the flow works

A typical implementation uses three actors: the client, the authentication endpoint, and the protected API.

- The client authenticates with primary credentials.
- The server issues a short-lived JWT access token and a refresh token.
- The client calls the API with the access token in the Authorization header.
- When the access token expires, the client sends the refresh token to the token endpoint.
- The server validates the refresh token, checks whether it is revoked or rotated, and returns a new access token.
- The old refresh token is invalidated if rotation is enabled.

A useful mental model is that the access token proves ?who you are right now? for API calls, while the refresh token proves ?you may ask for a new access token.? That distinction is important because the refresh token should not be accepted anywhere else.

The API should still validate every JWT locally. The refresh mechanism does not weaken JWT validation requirements; it only changes how clients recover from expiry. The practical result is fewer forced logins, not weaker authorization.

If your API is externally exposed, consider this pattern together with throttling controls on the refresh endpoint. ASP.NET Core Rate Limiting and Throttling in APIs is relevant because the refresh endpoint can become a target for credential stuffing, replay attempts, or noisy retry loops from misbehaving clients.

A practical environment you may recognize

A common case is an internal platform with a web frontend, a mobile app, and one or more service-to-service consumers. The frontend users expect to stay signed in during a workday, the mobile app may remain offline for periods of time, and the services need a predictable authentication model that does not depend on interactive login prompts.

In that environment, long-lived access tokens are usually a bad fit. If you stretch the access token lifetime to avoid frequent sign-ins, you also increase the window in which a stolen token remains useful. If you keep the access token short and provide refresh tokens, the client can renew silently while you retain the ability to revoke the session server-side.

This pattern also helps when user sessions need to survive short outages of the identity service or when frontends must avoid showing repeated login prompts after routine token expiry. The operational cost is that you now own refresh-token storage, rotation, revocation, and logging as part of the security boundary.

What this means in practice

In a real ASP.NET Core implementation, the access token is usually created as a signed JWT containing only the claims needed by downstream APIs. The refresh token is stored separately, often in a server-side table or cache, with metadata such as user ID, device or client identifier, issued time, expiry time, revocation status, and a hash of the token value rather than the raw token itself.

That storage decision is critical. If the refresh token database is compromised, raw token values become reusable credentials. Hashing the refresh token before persistence reduces the damage of a storage leak, similar to password-handling principles, although the exact hashing scheme should be chosen carefully and consistently.

On the validation side, the API still checks the JWT signature, issuer, audience, lifetime and relevant claims. The refresh endpoint checks a different set of controls:

- Is the refresh token present, valid and unexpired?
- Has it already been used if single-use rotation is enabled?
- Has it been revoked because the user signed out, changed password or an administrator invalidated the session?
- Does the presented token match the stored hash and belong to the expected user or client context?

These controls are what make the flow resilient. Without them, the refresh token becomes a long-lived bearer credential that is difficult to track and easy to abuse.

Implementation trade-offs

The main benefit of refresh tokens is usability without sacrificing short access-token lifetimes. The main cost is state. A purely stateless JWT model becomes partially stateful once you introduce refresh-token storage and revocation tracking.

That trade-off is usually acceptable, but it should be explicit. If your system is already built around distributed caches, user session records, or centralized identity services, the additional state is manageable. If your deployment model expects every authentication decision to be fully self-contained at the API layer, refresh tokens may introduce operational complexity you do not want.

Rotation is another trade-off. Refresh-token rotation reduces replay risk because each successful refresh invalidates the previous token. However, it also means you must handle concurrency carefully. Two near-simultaneous refresh attempts from the same client can cause one to fail if the earlier request already rotated the token. That is often acceptable, but mobile clients and unreliable networks can surface it more often than expected.

Revocation design also matters. If you need immediate session shutdown, you must keep some server-side state so the refresh token can be checked against a revoked list or session record. If you only need expiry-based control, you can simplify the implementation, but you give up fast invalidation.

Decision guidance

Use refresh tokens when the following are true:

- You need short-lived access tokens for API safety.
- Users or clients should remain signed in across token expiry.
- You can store refresh-token state securely on the server.
- You are prepared to implement rotation, revocation and audit logging.
- Your client can safely hold a refresh token, or you have a backend-for-frontend pattern that can protect it.

Avoid refresh tokens when the client cannot protect long-lived credentials, when session persistence is unnecessary, or when your threat model does not justify the extra state and lifecycle management. In some service-to-service cases, a client-credential flow or mutual TLS may be a better fit than refresh tokens.

A useful rule: if your only reason for using refresh tokens is to avoid re-authentication pain, but you cannot define secure storage and revocation rules, you do not yet have a production-ready design.

Common mistakes

The most common mistake is treating the refresh token like a second access token. It should not be sent to APIs, embedded in logs, or exposed to browser contexts without careful consideration. Its only job is to obtain new access tokens.

Another frequent error is making access tokens too long-lived because refresh tokens feel too complex. That shifts risk in the wrong direction. If access tokens last for many hours or days, revocation becomes harder and compromise impact increases.

Teams also underestimate client-side storage risk. If a single-page application stores refresh tokens in an unsafe browser location, an XSS issue can become a persistent account takeover. In browser-based systems, the storage model deserves as much scrutiny as the token logic itself.

Other mistakes include skipping rotation, failing to expire refresh tokens, not hashing tokens at rest, and not tracking which device or session each token belongs to. When incident response is required, opaque session records are far more useful than a flat list of anonymous token strings.

Compact production readiness checklist

Before production use, verify the following conditions:

- Access tokens have a short, intentional lifetime.
- Refresh tokens are stored server-side in hashed form, not as raw values.
- The refresh endpoint enforces authentication context, expiry and revocation checks.
- Rotation behavior is defined and tested for concurrent refresh attempts.
- Sign-out, password change and administrative revocation invalidate refresh tokens.
- Logs capture issuance, refresh, rotation failures and revocation events without exposing secrets.
- The client storage model matches the threat model for the platform type.
- Rate limiting or abuse controls exist on the refresh endpoint when appropriate.
- JWT validation still enforces signature, issuer, audience and claim checks.
- Recovery behavior is defined for expired, revoked and replayed refresh tokens.

What to verify before you ship

The implementation is only as strong as the surrounding controls. Before production use, confirm that the API rejects forged or expired JWTs, that refresh tokens cannot be reused after rotation if you have chosen one-time semantics, and that revocation is effective enough for your response requirements.

You should also validate operational behavior under failure. What happens if the token store is temporarily unavailable? Does the system fail closed or issue tokens without checking state? What happens if a client retries the refresh request during a network timeout? These questions are not edge cases; they are the conditions that reveal whether the design is safe under load and during incidents.

If your architecture already uses token-based APIs and identity claims, a refresh-token design can be a practical improvement rather than a complication. The key is to treat it as a stateful security subsystem, not just a convenience feature attached to JWTs. When you do that, the pattern gives you a balanced combination of API performance, user continuity and incident response control.

Use this guidance together with distributed SQL queries to connect the workflow with related operational context already available on the site.

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