ARTICLE

Secure .NET API Authentication with JWT and Refresh Tokens

JWT access tokens and refresh tokens solve different problems in .NET API authentication: one keeps requests efficient, the other reduces forced logouts. This article explains how the pair works, when to use it, the trade-offs to expect, and what to validate before production.

Programming - July 10, 2026

Key takeaways

JWT access tokens and refresh tokens are best understood as a two-token system. The access token authorizes API calls for a short period, while the refresh token lets a trusted client obtain a new access token without requiring the user to sign in again. That separation improves usability, but it also introduces a second credential that must be protected and managed carefully.

For .NET APIs, the practical question is not whether JWTs are ?secure? in isolation, but whether the authentication flow is designed with short-lived access tokens, refresh token storage, revocation, issuer validation, and safe rotation. If those controls are missing, the system may be convenient but still fragile under compromise or replay.

After reading this article, you should be able to decide whether this model fits your API, understand how the token pair works, apply a compact operational workflow to validate the design, and know what to check before moving it into production.

Why this matters operationally

A pure access-token design often creates one of two problems: access tokens are made long-lived to avoid reauthentication, which increases exposure if a token is stolen, or access tokens are made very short-lived with no renewal path, which leads to poor user experience and application churn. JWTs and refresh tokens address that tension by separating authorization from session continuity.

In a .NET API environment, that matters because service consumers, single-page apps, mobile apps, and internal tools often have different tolerance levels for reauthentication. A developer portal, a customer-facing SPA, and an internal admin console may all call the same API, but they do not all need the same session policy. The token model needs to reflect that operational reality rather than follow a generic template.

The trade-off is that refresh tokens become sensitive state. Unlike short-lived JWT access tokens, refresh tokens usually require persistence, rotation, and server-side tracking. That means they intersect with secret management, logging policy, exception handling, and incident response. If you are already hardening configuration storage, it is worth reviewing Secure .NET API Secrets with Azure Key Vault Integration so your token signing material and other secrets are handled consistently.

How JWT access tokens and refresh tokens work

A JWT access token is typically a signed bearer token presented to the API on each request. The API validates the signature, issuer, audience, lifetime, and claims before allowing the request to proceed. Its job is to make request-time authorization efficient and stateless from the API?s perspective.

A refresh token serves a different purpose. It is not usually sent to the API for normal authorization decisions. Instead, it is presented to a token endpoint when the access token expires. The authorization server verifies the refresh token, checks that it is still active, and issues a new access token. In many designs, it also rotates the refresh token and invalidates the previous one.

This arrangement lets you keep access tokens short-lived, which limits the impact of theft, while keeping the user or workload signed in for longer through the refresh mechanism. The critical design point is that the refresh token must be treated as an independent credential, not as a convenience string that can be copied into any store.

For APIs that enforce issuer and claims rules carefully, the authentication layer should behave consistently with the validation principles described in Securing ASP.NET Core APIs with JWT Authentication and Claims Validation. The refresh-token flow does not replace those checks; it depends on them.

Compact workflow

The workflow is intentionally narrow. It only works well if each token has a distinct purpose and each endpoint enforces its own checks. The access token should not be reused as a long-term session credential, and the refresh token should not be accepted as a general API authorization credential.

A practical scenario you may recognize

Consider a .NET-backed internal portal used by engineers, operations staff, and security analysts. The portal calls several APIs that return deployment status, audit events, and incident metadata. Users may keep the portal open all day, but they should not have to reauthenticate every 10 minutes. At the same time, the organization does not want a stolen browser session to stay valid for hours or days.

In this environment, short-lived JWT access tokens are a good fit for the API calls, while refresh tokens preserve continuity in the browser session. If the portal is fronted by a reverse proxy or multiple API tiers, the validation path still needs to be consistent across components so the token is accepted only when the issuer, audience, and claims match the intended trust boundary.

The same pattern also appears in mobile clients and field devices that need occasional connectivity. They benefit from fewer sign-ins, but the refresh token must be stored with platform-appropriate protections and the server must be able to revoke it if the device is lost or the account is compromised.

What this means in practice

In practice, JWT and refresh-token authentication is less about the token format and more about operational control. A short-lived JWT reduces exposure window, but the refresh token becomes the anchor for session longevity and therefore the part most likely to matter in an incident.

That changes how you think about logging, caching, and secret handling. Access tokens should not be written to logs, and refresh tokens should be handled with even more care because they can prolong access. If token refresh errors are turned into raw exception output, you may create diagnostic leakage or user-facing instability. Secure exception handling principles matter here as much as they do in any other security-sensitive API path; the goal is to return controlled failure responses without exposing internal details.

It also changes how you design service boundaries. If one API trusts another through service-to-service calls, the token exchange may not need a user refresh token at all; a client credentials flow or another machine identity pattern may be more appropriate. Use refresh tokens where you need long-lived user or device continuity, not as a default replacement for every authentication problem.

Decision guidance: when this approach fits

Use JWT access tokens plus refresh tokens when the client needs persistent sign-in state and you want short-lived bearer tokens at request time. That often includes browser-based applications, mobile clients, and operator tools where repeated authentication would create avoidable friction.

This approach is usually a poor fit when the client is a machine workload with no user context, when token persistence is difficult to secure, or when the system cannot support server-side refresh token tracking and revocation. In those cases, another identity flow may be simpler and safer.

A useful rule is this: if you cannot answer where the refresh token is stored, how it is protected, how it is rotated, and how it is revoked, you are not ready to rely on it in production.

Implementation trade-offs to evaluate

The main advantage of this model is a smaller blast radius for access-token theft. A short-lived JWT limits how long a stolen access token can be used, while the refresh token allows continuity without forcing users back through interactive login too often.

The main cost is complexity. You now need token issuance, token validation, refresh token storage, refresh token rotation, revocation logic, and operational visibility into failures. You also need to decide whether refresh tokens are scoped per device, per session, or per client, because that choice affects both user experience and incident response.

Token storage is another trade-off. Browser storage choices, mobile secure storage, and server-side session stores all carry different risk profiles. Refresh tokens should be protected with the same seriousness as other long-lived credentials. If your application already centralizes secrets and sensitive values, that architecture should extend to signing keys and related material rather than scattering them through configuration files.

Finally, revocation semantics are rarely free. If a refresh token can be revoked immediately, the server must maintain state and check it reliably. If revocation is delayed or approximate, the system becomes easier to scale but less precise during compromise response. Decide explicitly which behavior you need instead of assuming the implementation will provide it automatically.

Common mistakes

One common mistake is using long-lived JWT access tokens and calling that a refresh-token strategy. That only delays the problem and makes stolen tokens more dangerous.

Another mistake is failing to rotate refresh tokens. If the same refresh token can be reused indefinitely, replay risk increases and compromise detection becomes weaker.

A third mistake is storing refresh tokens in places that are too easy to exfiltrate, such as logs, URLs, browser-readable storage without a strong threat model, or unencrypted configuration.

Teams also often validate signature and expiry but ignore issuer, audience, or claims rules. That can lead to token acceptance across environments or clients that were never intended to trust one another. The validation rules need to be as specific as the API boundary itself.

Finally, many implementations handle refresh failures poorly. A failed refresh should trigger a controlled session recovery path, not an unbounded retry loop, a verbose stack trace, or a confusing half-authenticated state.

What to verify before production

Before shipping this flow, verify the following conditions in the actual deployment environment, not just in local development:

- Access tokens are short-lived and refresh tokens have an explicit lifetime.
- The API validates signature, issuer, audience, expiry, and required claims.
- Refresh tokens are stored securely and are not exposed in logs or URLs.
- Refresh token rotation is defined and old tokens are rejected as expected.
- Revocation behavior is documented for logout, password reset, device loss, and incident response.
- The token endpoint returns controlled errors without leaking sensitive details.
- Secrets and signing keys are managed outside application code and protected appropriately.
- The team has decided what happens when refresh fails: reauthentication, session termination, or step-up verification.

A compact readiness check is whether you can answer three questions with evidence: what token is accepted by which endpoint, what happens when a token is expired or revoked, and how quickly the system recovers without creating an insecure fallback.

Final takeaway

JWT access tokens and refresh tokens are a strong fit for .NET APIs when you need short-lived request authorization and longer-lived session continuity, but they are only secure when the surrounding controls are deliberate. Treat the access token as a short-lived bearer credential, treat the refresh token as a high-value secret, and verify validation, rotation, revocation, and storage before production. If those pieces are in place, the model gives you a practical balance of security and usability; if they are not, it mostly adds complexity.

Use this guidance together with partition pruning and prototype pollution to connect the workflow with related operational context already available on the site.

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