PuigFlows

JWT en Producción: Access Tokens, Refresh Tokens y Rotación con Detección de Robo

JWT in Production: Access Tokens, Refresh Tokens, and Rotation with Theft Detection

Premium

Guía · Intermedio · Proyectos Reales · ~35 min de lectura / ~35 min read
puigflows.com · Agosto 2026 / August 2026

ESPAÑOL

JWT en Producción: Access Tokens, Refresh Tokens y Rotación con Detección de Robo

El problema real: un token robado que no puedes revocar

Un JWT firmado es un cheque al portador. Quien lo tenga —tu frontend o un atacante que lo exfiltró con un script inyectado— obtiene acceso hasta que el token expire. Y aquí está la trampa: la mayoría de los tutoriales te enseñan a emitir tokens de larga duración sin ningún mecanismo de revocación, porque "los JWT son stateless". El resultado es un sistema donde cerrar sesión no cierra nada y donde un token filtrado sigue siendo válido durante días.

Esta guía monta el sistema completo que usan los proveedores de identidad serios: access tokens de vida corta, refresh tokens rotativos con detección de reuso, cookies bien configuradas, revocación real y verificación estricta. Todo con código que puedes llevar a producción.

La arquitectura: dos tokens con trabajos distintos

La pieza central es separar responsabilidades en dos credenciales:

Access token (JWT, 5–15 minutos). Viaja en cada petición a tu API. Se verifica con la firma, sin tocar la base de datos. Su vida corta es la defensa principal: si lo roban, la ventana de abuso se mide en minutos.

Refresh token (opaco, días o semanas). Solo sirve para pedir un access token nuevo. Es una cadena aleatoria sin estructura —no un JWT— que vive hasheada en tu base de datos. Como hay estado en el servidor, puedes revocarlo de verdad.

Esta división resuelve la tensión fundamental: la verificación sin estado te da latencia baja en cada petición, y el estado del refresh token te da control de sesión. Un access token de 10 minutos con refresh silencioso se siente, para el usuario, como una sesión infinita; para el atacante, como una puerta que se cierra sola.

Qué va dentro del JWT (y qué no)

El payload de un JWT va codificado en Base64URL, no cifrado: cualquiera que lo intercepte puede leerlo. Trátalo como una etiqueta con nombre, no como una caja fuerte. Nunca metas contraseñas, datos de tarjetas ni información personal sensible. Lo que sí necesita:

```
{
  "iss": "https://api.miapp.com",
  "aud": "miapp-web",
  "sub": "user_8231",
  "sid": "sess_f1c9",
  "iat": 1754380800,
  "exp": 1754381400,
  "jti": "0k3Xw9pQ"
}
```

`iss` y `aud` evitan que un token emitido para un servicio se acepte en otro. `sid` vincula el token a una sesión concreta (clave para revocar). `jti` es un identificador único que usaremos para la denylist. Si necesitas roles o permisos, inclúyelos, pero recuerda que quedan congelados hasta que el token expire: otro motivo para TTLs cortos.

Elección de algoritmo: ES256 antes que HS256

HS256 (HMAC) usa la misma clave para firmar y verificar. En cuanto tengas más de un servicio verificando tokens, esa clave simétrica acaba repartida por media infraestructura, y cualquier servicio que la tenga puede emitir tokens, no solo verificarlos. Con firmas asimétricas (ES256 o EdDSA), la clave privada vive solo en el servicio de autenticación y el resto verifica con la clave pública, publicada en un endpoint JWKS:

```
GET https://auth.miapp.com/.well-known/jwks.json
```

Incluye siempre el header `kid` (key ID) al firmar. Es lo que permite rotar claves sin downtime: publicas la clave nueva en el JWKS, firmas con ella, y los verificadores eligen la clave correcta por su `kid` mientras los tokens viejos terminan de expirar.

Dónde guardar los tokens en el navegador

En localStorage, no. Cualquier XSS —tuyo o de una dependencia de terceros— puede leerlo y exfiltrar los tokens en una línea. La opción correcta para aplicaciones web son cookies con estos atributos:

```
Set-Cookie: refresh_token=8fK2...;
  HttpOnly; Secure; SameSite=Strict;
  Path=/auth/refresh; Max-Age=1209600
```

`HttpOnly` la hace invisible para JavaScript. `Secure` la limita a HTTPS. `SameSite=Strict` corta el CSRF clásico. Y el detalle que casi todos omiten: `Path=/auth/refresh` hace que el refresh token solo viaje al endpoint de refresh, no en cada petición a tu API. El access token puedes mantenerlo en memoria (una variable, no storage) y renovarlo en silencio cuando expire.

Rotación de refresh tokens con detección de reuso

Aquí está el corazón de la guía. La rotación significa que cada refresh token es de un solo uso: al canjearlo, el servidor lo invalida y emite uno nuevo. Todos los tokens que descienden del login original forman una familia.

La detección de reuso es la consecuencia elegante: si alguien presenta un token que ya fue canjeado, hay dos poseedores del mismo token —el legítimo y el ladrón— y no sabes cuál es cuál. La respuesta correcta es revocar la familia completa y forzar un nuevo login. El robo de un refresh token pasa de ser una sesión secuestrada indefinidamente a una molestia de volver a iniciar sesión.

Implementación con FastAPI y SQL (los tokens se guardan hasheados: si te vuelcan la tabla, no obtienen credenciales utilizables):

```
import datetime as dt
import hashlib
import secrets

import jwt  # PyJWT

ACCESS_TTL = dt.timedelta(minutes=10)
REFRESH_TTL = dt.timedelta(days=14)

def _hash(token: str) -> str:
    return hashlib.sha256(token.encode()).hexdigest()

def issue_access_token(user_id: str, session_id: str) -> str:
    now = dt.datetime.now(dt.timezone.utc)
    payload = {
        "iss": "https://api.miapp.com",
        "aud": "miapp-web",
        "sub": user_id,
        "sid": session_id,
        "iat": now,
        "exp": now + ACCESS_TTL,
        "jti": secrets.token_urlsafe(8),
    }
    return jwt.encode(
        payload, PRIVATE_KEY, algorithm="ES256",
        headers={"kid": ACTIVE_KEY_ID},
    )

async def rotate_refresh_token(db, presented: str):
    row = await db.fetchrow(
        "SELECT * FROM refresh_tokens WHERE token_hash = $1",
        _hash(presented),
    )
    if row is None:
        raise AuthError(401, "invalid_token")

    if row["revoked_at"] is not None:
        # Reuso detectado: este token ya fue canjeado.
        # Alguien más tiene una copia. Se quema la familia entera.
        await db.execute(
            "UPDATE refresh_tokens SET revoked_at = now() "
            "WHERE family_id = $1 AND revoked_at IS NULL",
            row["family_id"],
        )
        raise AuthError(401, "reuse_detected")
```

```
    if row["expires_at"] < dt.datetime.now(dt.timezone.utc):
        raise AuthError(401, "expired_token")

    # Canje normal: invalidar el presentado, emitir el siguiente
    await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() WHERE id = $1",
        row["id"],
    )
    new_refresh = secrets.token_urlsafe(48)
    await db.execute(
        "INSERT INTO refresh_tokens "
        "(token_hash, family_id, user_id, expires_at) "
        "VALUES ($1, $2, $3, $4)",
        _hash(new_refresh), row["family_id"], row["user_id"],
        dt.datetime.now(dt.timezone.utc) + REFRESH_TTL,
    )
    access = issue_access_token(row["user_id"], str(row["family_id"]))
    return access, new_refresh
```

Dos matices de producción. Primero, la familia entera debe tener una caducidad absoluta (por ejemplo, 30 días desde el login): la rotación renueva tokens, no sesiones eternas. Segundo, concede unos segundos de gracia ante el canje concurrente —dos pestañas refrescando a la vez o un reintento de red— antes de declarar reuso, o generarás cierres de sesión fantasma que te costarán horas de soporte.

Verificación estricta del access token

La mayoría de los CVE históricos de JWT no son fallos criptográficos: son verificadores permisivos. La regla es fijar en el servidor todo lo que el token dice de sí mismo. En TypeScript con jose:

```
import { createRemoteJWKSet, jwtVerify } from "jose";

const jwks = createRemoteJWKSet(
  new URL("https://auth.miapp.com/.well-known/jwks.json"),
);

export async function verifyAccessToken(token: string) {
  const { payload } = await jwtVerify(token, jwks, {
    issuer: "https://api.miapp.com",
    audience: "miapp-web",
    algorithms: ["ES256"],  // fijado, nunca del header
    clockTolerance: 5,      // segundos de tolerancia de reloj
  });
  return payload;
}
```

Fijar `algorithms` neutraliza los ataques clásicos de confusión: el infame `alg: "none"` y el downgrade de RS256 a HS256, donde el atacante firma con la clave pública usada como secreto HMAC. Exigir `issuer` y `audience` evita aceptar tokens emitidos para otro contexto. La tolerancia de reloj debe ser de segundos; si necesitas minutos, arregla tu NTP, no tu verificador.

Logout y revocación real

Con la arquitectura anterior, cerrar sesión son dos operaciones: revocar la familia de refresh tokens (estado en tu base de datos) y neutralizar el access token vigente. Para lo segundo, una denylist en Redis con TTL igual al tiempo de vida restante del token:

```
async def logout(redis, db, payload, family_id):
    await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() "
        "WHERE family_id = $1 AND revoked_at IS NULL",
        family_id,
    )
    remaining = payload["exp"] - int(time.time())
    if remaining > 0:
        await redis.setex(f"deny:{payload['jti']}", remaining, "1")
```

El middleware consulta `deny:<jti>` en cada petición. Sí, eso reintroduce una lectura de estado, pero es una consulta a Redis de submilisegundo sobre un conjunto diminuto (solo tokens revocados sin expirar), y solo la pagas si necesitas logout instantáneo. Muchos sistemas aceptan la alternativa: sin denylist, un logout deja el access token vivo unos minutos. Decide explícitamente; no dejes que la decisión la tome el tutorial que copiaste.

El siguiente paso: tokens ligados al cliente (DPoP)

La rotación detecta el robo; DPoP (RFC 9449) lo vuelve inútil. El cliente genera un par de claves y, en cada petición, adjunta una prueba firmada (un JWT efímero con método HTTP, URL, timestamp y un `jti` único). El servidor de autorización liga los tokens a esa clave pública: un token robado sin la clave privada no sirve para nada.

```
POST /auth/token HTTP/1.1
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2Iiwiandr...
```

OAuth 2.1 recomienda tokens ligados al emisor para clientes públicos, y el ecosistema de agentes de IA (incluido MCP) empuja en la misma dirección: los agentes son exactamente el tipo de cliente público con tokens de larga vida donde el sender-constraining más aporta. Para una SPA en 2026 sigue siendo endurecimiento opcional —la rotación con detección de reuso cubre el caso común—, pero si diseñas hoy un sistema nuevo con clientes no confiables, evalúalo desde el principio.

Los errores que acaban en un incidente

Access tokens de horas o días. Anula la defensa principal del diseño. 5–15 minutos es el estándar; el refresh silencioso hace el resto.

Refresh tokens en claro en la base de datos. Un volcado de tabla se convierte en sesiones robadas. Hashea siempre (SHA-256 basta: son valores de alta entropía, no contraseñas).

Aceptar el algoritmo que declara el header. La lista de algoritmos la fija el verificador. Siempre.

No verificar aud e iss. Un token de tu entorno de staging o de otro servicio no debería abrir producción.

Tokens en localStorage. Cada dependencia de tu bundle es un vector XSS con acceso directo a las credenciales.

Logout que solo borra la cookie. Si no revocas en el servidor, la sesión sigue viva para quien tenga los tokens.

Rotación sin ventana de gracia. Dos pestañas concurrentes disparan falsos positivos de reuso y cierran sesiones legítimas.

Datos sensibles en el payload. Base64 no es cifrado; cualquier proxy o log intermedio puede leerlo.

Checklist de producción

Access token JWT de 5–15 minutos, firmado con ES256/EdDSA y header kid.

Refresh token opaco, hasheado en base de datos, con rotación en cada uso.

Detección de reuso que revoca la familia completa, con gracia para canjes concurrentes.

Caducidad absoluta de sesión independiente de la rotación.

Cookies HttpOnly; Secure; SameSite con Path acotado al endpoint de refresh.

Verificación con algoritmo, iss y aud fijados y tolerancia de reloj de segundos.

JWKS publicado y rotación de claves de firma probada, no solo documentada.

Logout que revoca refresh tokens y, si necesitas efecto inmediato, denylist por jti.

Métricas: refrescos por sesión, eventos de reuso detectado y latencia añadida por la denylist.

Nada de esto es exótico: es exactamente lo que hacen Auth0, Okta o cualquier proveedor de identidad debajo del capó. La diferencia entre "usamos JWT" y un sistema de autenticación defendible está en estos detalles, y ninguno de ellos requiere más que una tabla, un poco de Redis y verificadores estrictos.

¿Seguro que necesitas JWT? Sesiones clásicas y el patrón BFF

Antes de seguir profundizando conviene hacer la pregunta incómoda: si tu aplicación es un monolito con su propio frontend, una sesión clásica con cookie —un identificador aleatorio que apunta a estado en el servidor— resuelve el problema con menos piezas móviles. Revocación instantánea, sin rotaciones, sin JWKS. El JWT gana cuando hay múltiples servicios verificando identidad, APIs consumidas por terceros o un proveedor de identidad separado de tus APIs. Si no tienes ninguna de esas tres cosas, la complejidad de esta guía es opcional.

Para las SPA existe además un punto intermedio que la IETF ya recomienda formalmente: el patrón **BFF (Backend for Frontend)**. El borrador oficial de OAuth para aplicaciones de navegador (draft-ietf-oauth-browser-based-apps) es explícito: el navegador es un entorno hostil y, si puedes, no le des tokens en absoluto. Con BFF, un backend ligero actúa como cliente confidencial: guarda access y refresh tokens en el servidor y al navegador solo le entrega una cookie de sesión HttpOnly. Tu SPA habla con el BFF, el BFF adjunta el access token correcto a cada petición hacia las APIs, y no existe ningún token que un XSS pueda exfiltrar.

El coste es un salto de red adicional y un componente más que operar. La decisión práctica: si tu SPA ya tiene un backend propio (Next.js, Remix, un API gateway propio), el BFF es casi gratis y es la opción más segura. Si tu frontend es estático puro contra APIs de terceros, la arquitectura de tokens en el navegador de esta guía —cookies HttpOnly para el refresh token, access token en memoria— es el estándar razonable.

El esquema de base de datos completo

Todo el sistema de rotación descansa sobre una tabla. Esta es la definición completa con los índices que necesita en producción:

```
CREATE TABLE refresh_tokens (
  id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  token_hash  text NOT NULL UNIQUE,
  family_id   uuid NOT NULL,
  user_id     uuid NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  created_at  timestamptz NOT NULL DEFAULT now(),
  expires_at  timestamptz NOT NULL,
  revoked_at  timestamptz,
  -- caducidad absoluta de la familia (sesión)
  family_expires_at timestamptz NOT NULL,
  -- metadatos para la vista "mis sesiones"
  user_agent  text,
  ip_hash     text
);

-- El lookup principal: canje por hash
-- (UNIQUE ya crea el índice sobre token_hash)

-- Revocar una familia completa
CREATE INDEX idx_rt_family
  ON refresh_tokens (family_id)
  WHERE revoked_at IS NULL;

-- Listar sesiones activas de un usuario
CREATE INDEX idx_rt_user_active
```

```
  ON refresh_tokens (user_id)
  WHERE revoked_at IS NULL;
```

Los índices parciales con `WHERE revoked_at IS NULL` mantienen las estructuras pequeñas: solo indexan tokens vivos, que es lo único que consultas en caliente. La columna `family_expires_at` implementa la caducidad absoluta de la sesión de la que hablamos antes: se fija en el login y no se renueva jamás con las rotaciones.

La tabla crece con cada refresh, así que necesita limpieza. Un trabajo periódico con pg_cron basta:

```
-- Borra tokens expirados o revocados hace más de 30 días.
-- Se conservan un tiempo para auditoría e investigación
-- de incidentes, no se borran al momento.
SELECT cron.schedule(
  'purge-refresh-tokens',
  '15 4 * * *',
  $$
  DELETE FROM refresh_tokens
  WHERE expires_at < now() - interval '30 days'
     OR revoked_at < now() - interval '30 days'
  $$
);
```

¿Por qué conservar tokens revocados unas semanas? Porque cuando investigues un evento de reuso querrás reconstruir la cadena: qué token engendró a cuál, desde qué IP y con qué user agent. Si borras al instante, borras la evidencia.

Refresh silencioso en el frontend sin condiciones de carrera

El backend rota tokens; el frontend tiene que consumirlos sin que el usuario lo note. El patrón correcto tiene tres piezas: el access token vive en una variable de módulo (no en storage), las peticiones que reciben 401 disparan un refresh, y —la parte que casi todos implementan mal— los refreshes concurrentes se deduplican con un single-flight: si cinco peticiones fallan a la vez, solo una llama al endpoint de refresh y las otras cuatro esperan esa misma promesa.

```
// auth-client.ts
let accessToken: string | null = null;
let refreshing: Promise<string> | null = null;

async function refreshAccessToken(): Promise<string> {
  // single-flight: reutiliza el refresh en curso
  if (refreshing) return refreshing;

  refreshing = fetch("/auth/refresh", {
    method: "POST",
    credentials: "include",      // envía la cookie HttpOnly
```

```
    headers: { "X-Requested-With": "fetch" },
  })
    .then(async (res) => {
      if (!res.ok) throw new Error("refresh_failed");
      const body = await res.json();
      accessToken = body.access_token;
      return accessToken as string;
    })
    .finally(() => { refreshing = null; });

  return refreshing;
}

export async function apiFetch(path: string, init: RequestInit = {}) {
  const doFetch = (token: string | null) =>
    fetch(path, {
      ...init,
      headers: {
        ...init.headers,
        Authorization: token ? "Bearer " + token : "",
      },
    });

  let res = await doFetch(accessToken);
  if (res.status === 401) {
    try {
      const fresh = await refreshAccessToken();
      res = await doFetch(fresh);       // un único reintento
    } catch {
      redirectToLogin();
    }
  }
  return res;
}
```

Detalles que importan. El reintento tras el refresh es exactamente uno: si el segundo intento también devuelve 401, el problema no es el token y reintentar en bucle solo esconde el error. El `credentials: "include"` es lo que hace viajar la cookie del refresh token; recuerda que esa cookie tiene `Path=/auth/refresh`, así que ninguna otra petición la arrastra. Y si tu aplicación pasa mucho tiempo en segundo plano (pestañas dormidas), un listener de `visibilitychange` que refresque al despertar evita que la primera acción del usuario pague la latencia del 401 + refresh.

Una alternativa a reaccionar al 401 es el refresh proactivo: decodificar el `exp` del access token y renovarlo unos segundos antes de que caduque. Funciona, pero no elimina la necesidad del manejo del 401 (relojes desincronizados, revocaciones), así que impleméntalo como optimización, no como sustituto.

CSRF y el endpoint de refresh: los matices de SameSite

La cookie del refresh token con `SameSite=Strict` corta el CSRF clásico, pero conviene entender qué corta exactamente y qué no. SameSite gobierna cuándo el navegador adjunta la cookie en peticiones que cruzan sitios. Con Strict no viaja nunca desde otro sitio; con Lax viaja en navegaciones top-level con GET (hacer clic en un enlace), pero no en POST ni en subrecursos.

Para el endpoint de refresh, Strict es lo correcto: nadie navega "hacia" tu endpoint de refresh, así que no pierdes nada. Pero hay dos matices de producción. Primero, si tu frontend y tu API viven en dominios distintos (app.miapp.com y api.miapp.com son *same-site*; miapp.com y api-de-terceros.com no), las cookies SameSite no viajarán y necesitarás o unificar dominios o pasar al patrón BFF. Segundo, SameSite no sustituye una defensa explícita: un subdominio comprometido o un navegador antiguo pueden saltárselo. La defensa barata es exigir una cabecera personalizada:

```
@app.post("/auth/refresh")
async def refresh(request: Request):
    # Las peticiones cross-site de un formulario no pueden
    # añadir cabeceras personalizadas sin pasar por CORS.
    if request.headers.get("x-requested-with") != "fetch":
        raise HTTPException(403, "missing_csrf_header")
    ...
```

Un formulario HTML de otro origen no puede añadir `X-Requested-With`; solo puede hacerlo JavaScript del mismo origen (o de un origen que tu CORS permita explícitamente). Dos líneas, y el CSRF sobre el refresh queda cerrado incluso donde SameSite no llega.

Claves ES256 en la práctica: generación, JWKS y rotación

La teoría dice "firma con ES256 y publica un JWKS". La práctica empieza por generar el par de claves:

```
# Clave privada EC P-256 (la curva de ES256)
openssl ecparam -name prime256v1 -genkey -noout \
  -out jwt-signing-2026-08.pem

# Clave pública derivada
openssl ec -in jwt-signing-2026-08.pem -pubout \
  -out jwt-signing-2026-08.pub.pem
```

El nombre del fichero con fecha no es casual: el identificador de la clave (`kid`) puede ser esa misma etiqueta. El endpoint JWKS expone las claves públicas activas en el formato estándar:

```
from jwcrypto import jwk

@app.get("/.well-known/jwks.json")
async def jwks():
    keys = []
```

```
    for kid, pem in ACTIVE_PUBLIC_KEYS.items():
        key = jwk.JWK.from_pem(pem.encode())
        data = key.export_public(as_dict=True)
        data.update({"kid": kid, "use": "sig", "alg": "ES256"})
        keys.append(data)
    # Cache-Control corto: los verificadores refrescan pronto
    # cuando publiques una clave nueva
    return JSONResponse(
        {"keys": keys},
        headers={"Cache-Control": "public, max-age=300"},
    )
```

La rotación sin downtime es un procedimiento de cuatro pasos que conviene ensayar, no solo documentar:

1. Genera el par nuevo y publícalo en el JWKS *junto al* antiguo. Espera al menos el TTL de la caché del JWKS (el max-age de arriba más el margen de las librerías cliente).

2. Cambia la firma: los tokens nuevos salen con el `kid` nuevo. Los verificadores eligen clave por `kid`, así que ambas generaciones conviven.

3. Espera a que expire el último token firmado con la clave vieja (tu TTL de access token, con margen).

4. Retira la clave vieja del JWKS y destruye la privada.

Con access tokens de 10 minutos, el proceso completo cabe en media hora. Prográmalo como tarea trimestral: una rotación que solo has hecho en teoría es una rotación que fallará el día que la necesites de urgencia (una clave comprometida). Las librerías como jose en Node cachean el JWKS y lo refrescan solas al encontrar un `kid` desconocido; en Python, cachea el fetch con un TTL de minutos y invalida ante `kid` no encontrado.

Sesiones multi-dispositivo: la vista "mis sesiones"

Cada login crea una familia de refresh tokens, y eso te da gratis algo que los usuarios esperan: la pantalla de sesiones activas con su botón de "cerrar esta sesión" y "cerrar todas las demás". La consulta es directa gracias a los metadatos que guardamos por familia:

```
-- Sesiones activas: el token vivo más reciente de cada familia
SELECT DISTINCT ON (family_id)
       family_id,
       created_at AS last_refresh,
       user_agent,
       ip_hash
FROM refresh_tokens
WHERE user_id = $1
  AND revoked_at IS NULL
  AND expires_at > now()
ORDER BY family_id, created_at DESC;
```

```
@app.delete("/auth/sessions/{family_id}")
async def revoke_session(family_id: str, user=Depends(current_user)):
    result = await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() "
        "WHERE family_id = $1 AND user_id = $2 "
        "AND revoked_at IS NULL",
        family_id, user.id,
    )
    # El access token de esa sesión sigue vivo unos minutos
    # salvo que además lo metas en la denylist por sid.
    return {"revoked": True}
```

Fíjate en el filtro por `user_id` además de `family_id`: sin él, cualquier usuario autenticado podría revocar sesiones ajenas adivinando UUIDs. Y el comentario final importa: revocar la familia corta los refreshes futuros, pero el access token en vuelo sigue siendo válido hasta expirar. Si tu producto promete "cerrar sesión en otro dispositivo *ya*", añade el `sid` de esa familia a la denylist de Redis igual que en el logout.

El caso especial que no debes olvidar: el cambio de contraseña. La expectativa universal del usuario es que cambiar la contraseña expulse a cualquiera que estuviera dentro. Revoca todas las familias del usuario excepto la de la sesión actual, y añade a la denylist los `sid` del resto.

Microservicios: dónde verificar y cómo propagar identidad

Con varios servicios detrás de un gateway hay dos modelos. El primero: cada servicio verifica el JWT completo (firma contra JWKS, iss, aud, exp). Es lo más robusto —ningún servicio confía en nada que no pueda comprobar— y el coste es mínimo porque la clave pública se cachea y la verificación ES256 tarda microsegundos. El segundo: el gateway verifica una vez y pasa la identidad hacia dentro en una cabecera interna. Ahorra algo de CPU, pero convierte tu red interna en una zona de confianza: cualquier cosa que pueda hablar con un servicio interno puede falsificar la cabecera. Solo es aceptable si la red interna tiene mTLS o políticas de malla que garanticen que la cabecera solo puede venir del gateway.

La recomendación práctica es la primera: verificación completa en cada borde, y si el gateway ya verificó, que los servicios lo repitan de todos modos. Defensa en profundidad barata.

Cuando un servicio llama a otro *en nombre del usuario*, resiste la tentación de reenviar el access token original tal cual. Ese token tiene una audiencia concreta (`aud: "miapp-web"`), y si el servicio B lo acepta, has vuelto a los tokens multiuso que `aud` intentaba evitar. El mecanismo estándar es el **token exchange** (RFC 8693): el servicio A presenta el token del usuario al servidor de autorización y recibe un token nuevo con la audiencia del servicio B, conservando el `sub` original y añadiendo la cadena de delegación en el claim `act`. Para llamadas entre servicios sin usuario (trabajos programados, colas), el flujo correcto es client_credentials con su propia identidad de servicio, nunca un token de usuario reciclado.

Cuando Redis se cae: fail-open o fail-closed

La denylist introduce una dependencia en el camino caliente, y las dependencias fallan. Tienes que decidir por adelantado qué pasa con las verificaciones cuando Redis no responde. **Fail-open** (si Redis no contesta, el token se acepta) prioriza disponibilidad: un logout puede tardar unos minutos en hacerse

efectivo durante el incidente, pero tu API sigue sirviendo. **Fail-closed** (si Redis no contesta, 401 para todos) prioriza seguridad y tumba la plataforma entera con la caché.

Para la mayoría de las aplicaciones, fail-open con TTLs cortos es la respuesta defendible: la ventana de riesgo está acotada por el TTL del access token (minutos) y solo afecta a tokens que ya fueron revocados explícitamente durante el incidente. Impleméntalo con un timeout agresivo y métricas:

```
async def is_denylisted(redis, jti: str) -> bool:
    try:
        return await asyncio.wait_for(
            redis.exists("deny:" + jti), timeout=0.05
        ) == 1
    except (asyncio.TimeoutError, RedisError):
        DENYLIST_CHECK_FAILURES.inc()   # métrica Prometheus
        return False   # fail-open: decisión explícita
```

Si manejas operaciones de alto riesgo (transferencias, cambios de credenciales), aplica fail-closed selectivo solo en esos endpoints. La granularidad es tu amiga: no toda la API necesita la misma postura.

Observabilidad y testing del sistema de autenticación

Un sistema de tokens sin métricas es una caja negra que solo entiendes durante el incidente. Lo mínimo que quieres exportar: refreshes por resultado (éxito, expirado, reuso detectado), verificaciones fallidas por motivo (firma, exp, aud) y la latencia de la denylist.

```
from prometheus_client import Counter

TOKEN_REFRESH = Counter(
    "auth_token_refresh_total",
    "Canjes de refresh token",
    ["result"],  # success | expired | reuse_detected
)
TOKEN_VERIFY_FAIL = Counter(
    "auth_token_verify_failures_total",
    "Verificaciones de access token fallidas",
    ["reason"],  # bad_signature | expired | bad_aud | denylisted
)
```

Dos alertas valen especialmente la pena. Un pico de `reuse_detected` por encima del ruido de base (los falsos positivos de pestañas concurrentes deberían rondar cero con la ventana de gracia) huele a robo de tokens o a un bug del cliente; ambos merecen atención inmediata. Y un pico de `bad_signature` justo después de una rotación de claves significa que algún verificador no refrescó el JWKS: tienes un despliegue con la caché envenenada.

En cuanto a tests, la lógica de rotación es exactamente el tipo de código que merece una batería exhaustiva, porque sus fallos son silenciosos hasta que son catastróficos:

```
async def test_reuse_detection_burns_family(db):
    t1 = await login(db, user_id="u1")          # familia nueva
    t2 = await rotate(db, t1)                   # canje normal
    with pytest.raises(AuthError) as e:
        await rotate(db, t1)                    # reuso de t1
    assert e.value.code == "reuse_detected"
    # t2 también debe haber muerto con la familia
    with pytest.raises(AuthError):
        await rotate(db, t2)

async def test_concurrent_refresh_grace(db):
    t1 = await login(db, user_id="u1")
    # dos pestañas canjean el mismo token casi a la vez
    r1, r2 = await asyncio.gather(
        rotate(db, t1), rotate(db, t1),
        return_exceptions=True,
    )
    # dentro de la ventana de gracia: ninguna sesión muere
    assert not all(isinstance(r, AuthError) for r in (r1, r2))

def test_verifier_rejects_none_alg():
    forged = make_token_with_alg_none(sub="u1")
    with pytest.raises(InvalidTokenError):
        verify_access_token(forged)
```

El tercer test parece paranoico hasta que recuerdas que `alg: "none"` fue un CVE real en múltiples librerías. Verificar que tu verificador rechaza lo que debe rechazar es tan importante como verificar que acepta lo válido. Añade casos para tokens expirados con distintos desfases de reloj, audiencia incorrecta y `kid` desconocido, y tendrás la parte de tu suite con mejor relación coste/beneficio.

Preguntas frecuentes

¿Puedo usar JWT como cookie de sesión y saltarme los refresh tokens? Puedes, pero heredas lo peor de ambos mundos: o el JWT dura poco y el usuario se desloguea cada 15 minutos, o dura mucho y no puedes revocarlo. El par access + refresh existe precisamente para no elegir entre esas dos.

¿Por qué no cifrar el payload con JWE en lugar de solo firmarlo? Porque casi nunca es el problema real. Si hay datos que el cliente no debe leer, la solución simple es no meterlos en el token: guarda una referencia (`sub`, `sid`) y resuelve el resto en el servidor. JWE añade gestión de claves de cifrado a cambio de ocultar algo que no deberías estar enviando.

¿SHA-256 sin salt para hashear refresh tokens no es débil? No para este caso. El salt y los algoritmos lentos (bcrypt, argon2) existen porque las contraseñas humanas son adivinables por fuerza bruta. Un refresh token de 48 bytes aleatorios tiene ~384 bits de entropía: no hay diccionario ni rainbow table posible. SHA-256 simple es correcto y te permite buscar por igualdad exacta con un índice normal.

¿Qué TTL pongo a la familia (caducidad absoluta)? Depende del riesgo del producto. 30 días es razonable para un SaaS general; banca u operaciones sensibles bajan a horas o a un día. La pregunta que te lo decide: ¿cuánto tiempo es aceptable que un dispositivo robado y desbloqueado siga con la sesión abierta si el usuario no hace nada?

¿Access token en cookie o en cabecera Authorization? Para tu propio frontend web, una cookie HttpOnly con SameSite también para el access token es defendible (y elimina el token de la memoria de JavaScript). La cabecera gana cuando tu API la consumen clientes no-navegador (móvil, CLIs, terceros) y cuando quieres evitar el CSRF por diseño. Muchos sistemas maduros acaban soportando ambas.

¿Cada cuánto roto las claves de firma? Sin requisito de cumplimiento que diga otra cosa, cada 90 días es un buen equilibrio: suficiente para acotar el daño de una filtración no detectada y suficiente práctica para que el procedimiento nunca se oxide.

Caso práctico: el flujo completo, de login a logout

Las piezas anteriores —emisión, rotación, cookies, denylist— cobran sentido cuando las ves ensambladas. Este es el ciclo de vida completo de una sesión en FastAPI, empezando por el login:

```
@app.post("/auth/login")
async def login(creds: LoginRequest, request: Request, response: Response):
    user = await verify_credentials(creds.email, creds.password)
    if user is None:
        # Mismo mensaje y tiempo de respuesta exista o no el
        # email: no filtres qué cuentas existen.
        raise HTTPException(401, "invalid_credentials")

    # Nueva familia = nueva sesión
    family_id = uuid.uuid4()
    refresh = secrets.token_urlsafe(48)
    now = dt.datetime.now(dt.timezone.utc)
    await db.execute(
        "INSERT INTO refresh_tokens "
        "(token_hash, family_id, user_id, expires_at, "
        " family_expires_at, user_agent, ip_hash) "
        "VALUES ($1, $2, $3, $4, $5, $6, $7)",
        _hash(refresh), family_id, user.id,
        now + REFRESH_TTL,
        now + dt.timedelta(days=30),          # caducidad absoluta
        request.headers.get("user-agent", "")[:300],
        _hash(request.client.host),
    )

    response.set_cookie(
        key="refresh_token",
        value=refresh,
        httponly=True,
```

```
        secure=True,
        samesite="strict",
        path="/auth/refresh",
        max_age=int(REFRESH_TTL.total_seconds()),
    )
    access = issue_access_token(str(user.id), str(family_id))
    return {"access_token": access, "expires_in": 600}
```

Tres decisiones silenciosas que merecen comentario. La respuesta ante credenciales incorrectas es idéntica exista o no la cuenta, incluida la latencia: comparar contra un hash fantasma cuando el email no existe evita que el tiempo de respuesta delate qué cuentas hay registradas. La IP se guarda hasheada: te sirve para correlacionar en una investigación sin acumular datos personales en claro. Y el access token vuelve en el cuerpo de la respuesta, no en una cookie, porque este diseño lo mantiene en memoria del cliente.

El endpoint de refresh une la rotación con la cookie:

```
@app.post("/auth/refresh")
async def refresh_endpoint(request: Request, response: Response):
    if request.headers.get("x-requested-with") != "fetch":
        raise HTTPException(403, "missing_csrf_header")

    presented = request.cookies.get("refresh_token")
    if not presented:
        raise HTTPException(401, "no_refresh_token")

    try:
        access, new_refresh = await rotate_refresh_token(db, presented)
    except AuthError as e:
        # Ante cualquier fallo, limpia la cookie: el cliente
        # sabrá que toca login completo.
        response.delete_cookie("refresh_token", path="/auth/refresh")
        raise HTTPException(401, e.code)

    response.set_cookie(
        key="refresh_token", value=new_refresh,
        httponly=True, secure=True, samesite="strict",
        path="/auth/refresh",
        max_age=int(REFRESH_TTL.total_seconds()),
    )
    return {"access_token": access, "expires_in": 600}
```

Y el logout revoca en servidor antes de limpiar en cliente:

```
@app.post("/auth/logout")
async def logout_endpoint(request: Request, response: Response,
```

```
                           payload=Depends(verify_access)):
    await logout(redis, db, payload, payload["sid"])
    response.delete_cookie("refresh_token", path="/auth/refresh")
    return {"ok": True}
```

Con esto, el recorrido completo queda cerrado: el login crea la familia y siembra las dos credenciales; cada refresh quema un eslabón y forja el siguiente; el logout mata la familia y neutraliza el access token vigente. Cualquier token que aparezca fuera de ese carril —un refresh ya canjeado, un access en la denylist— es ruido hostil que el sistema rechaza solo.

Clientes móviles y nativos: sin cookies, mismo diseño

En iOS y Android no hay cookies que gestionar ni XSS del que esconderse, pero el problema del almacenamiento reaparece con otra cara: un dispositivo puede perderse, rootearse o compartirse. La regla es guardar el refresh token en el almacén seguro del sistema —Keychain en iOS, Keystore/EncryptedSharedPreferences en Android— y nunca en preferencias en claro, ficheros o bases de datos locales.

```
// Android (Kotlin): EncryptedSharedPreferences
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()

val securePrefs = EncryptedSharedPreferences.create(
    context, "auth_store", masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
securePrefs.edit()
    .putString("refresh_token", newRefreshToken)
    .apply()
```

El resto del diseño se conserva intacto: el access token vive en memoria del proceso, el refresh token rota en cada uso y la detección de reuso funciona exactamente igual. Los matices móviles son operativos. Primero, el refresh en segundo plano: si el sistema suspende tu app durante días, el refresh token puede expirar; maneja el caso con un login suave (biometría contra el Keychain) en lugar de plantar al usuario en el formulario de contraseña. Segundo, los reintentos de red agresivos del móvil hacen más probable el canje concurrente: la ventana de gracia del servidor deja de ser opcional. Tercero, si tu app tiene widget o extensiones, cada proceso extra es un consumidor más del mismo refresh token; considera familias separadas por superficie para poder revocarlas de forma independiente.

Rate limiting en los endpoints de autenticación

Los endpoints de auth son los más atacados de cualquier API: el login recibe credential stuffing (listas de contraseñas filtradas probadas en masa) y el refresh recibe fuerza bruta contra tokens. El rate limiting aquí no es el genérico de tu API; necesita dimensiones propias.

Para el login, limita por dos claves a la vez: por IP (frena el escaneo masivo desde un origen) y por cuenta objetivo (frena el ataque distribuido contra un usuario concreto). Los umbrales razonables difieren: una IP legítima detrás de un NAT corporativo puede generar decenas de logins por minuto; una cuenta individual, no.

```
async def check_login_limits(redis, email: str, ip: str):
    # Por IP: 30 intentos / 5 min (NAT-friendly)
    ip_key = "rl:login:ip:" + _hash(ip)
    # Por cuenta: 5 intentos / 5 min
    acct_key = "rl:login:acct:" + _hash(email.lower())

    async with redis.pipeline() as pipe:
        pipe.incr(ip_key);  pipe.expire(ip_key, 300, nx=True)
        pipe.incr(acct_key); pipe.expire(acct_key, 300, nx=True)
        ip_n, _, acct_n, _ = await pipe.execute()

    if ip_n > 30 or acct_n > 5:
        # 429 con Retry-After, siempre el mismo cuerpo
        raise HTTPException(
            429, "too_many_attempts",
            headers={"Retry-After": "300"},
        )
```

El límite por cuenta convierte además el bloqueo en señal: si una cuenta alcanza su límite repetidamente sin login exitoso, avisa al dueño ("hemos detectado intentos de acceso") y considera exigir un segundo factor en el siguiente login correcto. Para el endpoint de refresh el patrón es distinto: un cliente sano refresca como mucho unas pocas veces por minuto por sesión, así que un límite por familia (no por IP) de, digamos, 10 por minuto atrapa clientes rotos y ataques por igual sin castigar a nadie legítimo.

Roles y permisos dentro del token: frescura contra latencia

Meter roles en el JWT (`"role": "admin"`) es cómodo: la autorización se resuelve sin tocar la base de datos. Pero los claims quedan congelados hasta que el token expira, y eso convierte cada decisión de autorización en una pregunta sobre frescura: si degradas a un administrador malicioso, ¿cuántos minutos de administrador le quedan? Con access tokens de 10 minutos, la respuesta es "hasta 10", y para la mayoría de los productos eso es aceptable. Para operaciones destructivas no lo es.

El patrón maduro es de dos niveles. Las comprobaciones baratas y frecuentes (¿puede ver esta página?, ¿puede llamar a este endpoint de lectura?) usan los claims del token. Las operaciones sensibles (borrar el proyecto, cambiar permisos de otros, exportar datos) revalidan contra la base de datos en el momento, ignorando lo que diga el token. Así pagas la consulta solo donde la frescura importa.

Vigila también el tamaño. Cada claim viaja en cada petición, y los proxies intermedios suelen limitar las cabeceras a 8 KB; un token con listas largas de permisos o pertenencias a grupos se acerca peligrosamente. Si tu modelo de permisos no cabe en un puñado de claims, el token debe llevar la referencia (`sub`, `sid`, quizá un `role` grueso) y el detalle debe vivir en el servidor, cacheado si hace falta. Un JWT no es una base de datos portátil.

Anatomía de un incidente: cómo se usa todo esto bajo fuego

Para aterrizar el valor del diseño, un incidente plausible de principio a fin. Lunes, 10:40: la alerta de `reuse_detected` pasa de su ruido de base (cero o uno al día) a catorce eventos en una hora. No es un despliegue nuevo ni un bug conocido del cliente.

10:55: con la tabla de tokens y sus metadatos, el equipo reconstruye las cadenas afectadas. Las catorce familias comparten patrón: el canje legítimo venía de las IPs y user agents habituales de cada usuario; el segundo canje —el que disparó la detección— salía de un mismo rango de IPs de un hosting barato. Conclusión provisional: alguien está exfiltrando refresh tokens de un subconjunto de usuarios y canjeándolos desde su infraestructura.

11:10: contención. Las catorce familias ya se revocaron solas (eso es la detección de reuso funcionando); el equipo revoca además todas las familias de esos usuarios, mete sus `sid` en la denylist y fuerza re-login. El daño por usuario queda acotado a los minutos de vida del último access token robado.

11:30: ¿por dónde salieron los tokens? Los usuarios afectados comparten versión de la app y todos habían instalado una extensión de navegador concreta. No es un XSS del sitio: es una extensión maliciosa leyendo el almacenamiento de la página. Y aquí llega el dividendo del diseño: como el refresh token vive en una cookie HttpOnly con Path acotado y el access token en memoria, la extensión solo capturó access tokens de minutos de vida. Sin rotación ni cookies bien puestas, ese mismo incidente habría sido "sesiones robadas indefinidamente hasta que alguien se dé cuenta".

12:00: cierre. Rotación de claves de firma por precaución (el procedimiento ensayado del apartado anterior, sin downtime), aviso a los usuarios afectados, y una regla nueva de alerta: canjes de refresh desde ASNs de hosting cuando la familia nació en una IP residencial. El post-mortem no propone rediseñar nada: propone bajar el TTL del access token de 10 a 5 minutos. El sistema funcionó.

Migraciones: llegar a este diseño desde donde estás

Pocas veces se construye esto desde cero; lo normal es migrar desde algo. Dos rutas habituales.

De HS256 a ES256 sin cerrar sesiones. El truco es que verificar y emitir son operaciones independientes. Paso uno: los verificadores aceptan ambos algoritmos, cada uno con su clave, eligiendo por `kid` (los tokens HS256 viejos no llevan `kid`, lo que ya los identifica). Paso dos: el emisor cambia a ES256. Paso tres: cuando el último token HS256 ha expirado —un TTL de access token después—, se elimina el soporte HS256 del verificador. Total: tres despliegues y ningún usuario deslogueado.

```
def verify_transitional(token: str):
    header = jwt.get_unverified_header(token)
    if "kid" in header:                    # generación nueva
        return jwt.decode(
            token, public_key_for(header["kid"]),
            algorithms=["ES256"],
            audience="miapp-web", issuer=ISS,
        )
    # generación vieja, en extinción
    LEGACY_HS256_VERIFIED.inc()            # métrica: debe tender a 0
    return jwt.decode(
```

```
        token, LEGACY_SECRET, algorithms=["HS256"],
        audience="miapp-web", issuer=ISS,
    )
```

La métrica del camino legado es la clave operativa: cuando lleva unos días plana en cero, puedes borrar la rama con confianza en lugar de con fe.

De sesiones clásicas a access + refresh. No migres a los usuarios en masa: deja que la migración ocurra sola. El middleware acepta temporalmente ambas credenciales (cookie de sesión vieja o Bearer nuevo). Los logins nuevos emiten el par de tokens; las sesiones viejas siguen funcionando hasta que expiran por su propio curso. En unas semanas, el tráfico con sesión clásica se agota y retiras el código. La alternativa —invalidar todas las sesiones un martes y forzar login global— es técnicamente más simple y operativamente peor: los picos de soporte no compensan.

Los errores de librería que conviene conocer por nombre

La historia de JWT está llena de vulnerabilidades que no eran del estándar sino de implementaciones concretas, y conocerlas te vacuna contra sus patrones.

El clásico `alg: "none"` (CVE-2015-9235 en node-jsonwebtoken, entre otros) permitía presentar tokens sin firma que algunas librerías aceptaban alegremente. La lección permanente: el verificador fija la lista de algoritmos, jamás la lee del token. La confusión RS256/HS256 —firmar con la clave pública usada como secreto HMAC— reapareció durante años en librerías distintas; en 2024 le tocó a python-jose (CVE-2024-33663, confusión de algoritmos con claves ECDSA). Y el error eterno a nivel de aplicación: llamar a la función de decodificación sin verificación —`jwt.decode` con la verificación desactivada, o el `decode()` de jsonwebtoken que no verifica nada, frente a `verify()` que sí— y usar esos claims para autorizar. Audita tu código buscando esas llamadas: es un grep de un minuto que ha encontrado agujeros en más de una empresa.

En el ecosistema Python actual, PyJWT es la opción con mantenimiento más activo y una API que te obliga a pasar `algorithms` explícitamente. En Node, jose es la referencia moderna (soporta JWKS remoto, EdDSA y DPoP) frente al veterano jsonwebtoken. Sea cual sea la elección, fija la versión, suscríbete a sus avisos de seguridad y trata cualquier actualización del verificador como un cambio sensible que pasa por revisión.

Glosario rápido

Access token. Credencial de vida corta (minutos) que autoriza peticiones a la API; aquí, un JWT verificable sin estado.

Refresh token. Credencial opaca de vida larga cuyo único poder es obtener nuevos access tokens; vive hasheada en el servidor.

Familia. El linaje completo de refresh tokens descendiente de un login; la unidad de revocación de una sesión.

Rotación. Política por la que cada refresh token es de un solo uso y su canje emite el siguiente.

Detección de reuso. Alarma que salta cuando se presenta un token ya canjeado; implica dos poseedores y quema la familia.

JWKS. Documento JSON con las claves públicas de verificación, publicado en una URL conocida.

kid. Identificador de clave en el header del JWT; permite convivir varias claves y rotarlas sin downtime.

Denylist. Lista de tokens revocados antes de su expiración, consultada en cada verificación; en este diseño, un conjunto pequeño en Redis con TTL.

DPoP. Mecanismo (RFC 9449) que liga los tokens a una clave del cliente, volviendo inútil el token robado sin esa clave.

BFF. Backend for Frontend: backend ligero que guarda los tokens del lado servidor y le da al navegador solo una cookie de sesión.

Tokens de un solo uso: reset de contraseña, magic links y cambio de email

Alrededor del sistema de sesiones orbitan otros tokens con pinta parecida y requisitos opuestos: el enlace de "restablecer contraseña", el magic link de login sin contraseña, la confirmación de cambio de email. La tentación es resolverlos con JWTs firmados —"si ya tengo la infraestructura..."— y es exactamente la herramienta equivocada. Estos tokens deben ser de un solo uso, y un JWT firmado no puede serlo sin estado en el servidor: firmado una vez, vale hasta que expira, lo canjeen una vez o veinte.

La forma correcta es la misma que la de los refresh tokens: valor opaco aleatorio, hasheado en base de datos, con expiración corta y una columna que registra el canje.

```
CREATE TABLE one_time_tokens (
  token_hash  text PRIMARY KEY,
  purpose     text NOT NULL,   -- password_reset | magic_link | email_change
  user_id     uuid NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  payload     jsonb,           -- p. ej. el email nuevo propuesto
  expires_at  timestamptz NOT NULL,
  used_at     timestamptz
);

-- Canje atómico: solo gana una petición aunque lleguen dos
UPDATE one_time_tokens
SET used_at = now()
WHERE token_hash = $1
  AND purpose = $2
  AND used_at IS NULL
  AND expires_at > now()
RETURNING user_id, payload;
```

El UPDATE condicional con RETURNING es el detalle que evita la carrera: si dos peticiones canjean el mismo enlace a la vez, solo una recibe fila; la otra recibe cero filas y un error limpio. Las expiraciones deben ser cortas de verdad: 15-30 minutos para un reset de contraseña, 5-10 para un magic link. Y el canje de un reset de contraseña debe arrastrar consecuencias en el sistema de sesiones que ya

montamos: contraseña nueva significa revocar todas las familias del usuario y vaciar sus sesiones, porque el motivo más común de un reset es precisamente la sospecha de compromiso.

El cambio de email merece su coreografía propia, porque es el objetivo favorito de quien secuestra una sesión (cambiando el email, el atacante hereda los futuros resets). El patrón robusto es en dos fases: el enlace de confirmación llega al email *nuevo* (prueba de posesión), pero el aviso con opción de cancelación llega al email *viejo*, con su propio token de un solo uso que revierte el cambio y revoca las sesiones. Esa ventana de reversión es la diferencia entre un susto y una cuenta perdida.

El coste real: números de latencia para decidir con datos

Todas las decisiones de esta guía (¿stateless o denylist?, ¿ES256 o HS256?, ¿verificar en cada servicio?) son en el fondo decisiones de presupuesto de latencia, así que cierra la guía con los órdenes de magnitud que las gobiernan. Verificar una firma HS256 cuesta en torno a un microsegundo; una ES256, decenas de microsegundos (y firmar ES256 es más barato que verificarlo). Una consulta a Redis en la misma red, medio milisegundo; una consulta indexada a PostgreSQL, uno o dos milisegundos bajo carga normal.

Léelos juntos y las conclusiones se caen solas. La diferencia entre HS256 y ES256 es irrelevante frente a cualquier I/O: elegir algoritmo por rendimiento es optimizar el 0,1% del presupuesto; elige por seguridad operativa (la clave privada en un solo sitio), que es donde está la diferencia real. Verificar el JWT en cada microservicio añade microsegundos por salto: la defensa en profundidad es gratis a efectos prácticos. La denylist en Redis añade ~0,5 ms por petición: perceptible en un p99 ajustado, asumible casi siempre, y con la política de fail-open el día malo de Redis no se convierte en el día malo de tu API. Y la alternativa de sesiones clásicas —una consulta por petición— cuesta uno o dos milisegundos más una dependencia dura de la base de datos en cada endpoint: perfectamente válida para un monolito, cara para veinte servicios.

La moraleja de toda la guía cabe en una frase: los tokens no se eligen por moda sino por presupuesto —de latencia, de operación y de riesgo—, y un sistema access + refresh bien montado es la forma de pagar poco de los tres a la vez.

ENGLISH

JWT in Production: Access Tokens, Refresh Tokens, and Rotation with Theft Detection

The real problem: a stolen token you can't revoke

A signed JWT is a bearer check. Whoever holds it —your frontend, or an attacker who exfiltrated it with an injected script— gets access until the token expires. And here's the trap: most tutorials teach you to issue long-lived tokens with no revocation mechanism at all, because "JWTs are stateless". The result is a system where logging out closes nothing, and where a leaked token stays valid for days.

This guide builds the complete system that serious identity providers use: short-lived access tokens, rotating refresh tokens with reuse detection, properly configured cookies, real revocation, and strict verification. All with code you can take to production.

The architecture: two tokens with different jobs

The central piece is splitting responsibilities across two credentials:

Access token (JWT, 5–15 minutes). Travels on every request to your API. It's verified by signature, without touching the database. Its short life is the primary defense: if it's stolen, the abuse window is measured in minutes.

Refresh token (opaque, days or weeks). Only good for requesting a new access token. It's a random string with no structure —not a JWT— that lives hashed in your database. Because there's server-side state, you can actually revoke it.

This split resolves the fundamental tension: stateless verification gives you low latency on every request, and refresh-token state gives you session control. A 10-minute access token with silent refresh feels, to the user, like an infinite session; to the attacker, like a door that closes on its own.

What goes inside the JWT (and what doesn't)

A JWT payload is Base64URL-encoded, not encrypted: anyone who intercepts it can read it. Treat it like a name tag, not a safe. Never put passwords, card data, or sensitive personal information in it. What it does need:

```
{
  "iss": "https://api.myapp.com",
  "aud": "myapp-web",
  "sub": "user_8231",
  "sid": "sess_f1c9",
  "iat": 1754380800,
  "exp": 1754381400,
  "jti": "0k3Xw9pQ"
}
```

`iss` and `aud` prevent a token issued for one service from being accepted by another. `sid` ties the token to a specific session (key for revocation). `jti` is a unique identifier we'll use for the denylist. If you need roles or permissions, include them, but remember they're frozen until the token expires: another reason for short TTLs.

Algorithm choice: ES256 over HS256

HS256 (HMAC) uses the same key to sign and verify. As soon as more than one service verifies tokens, that symmetric key ends up scattered across half your infrastructure, and any service holding it can mint tokens, not just verify them. With asymmetric signatures (ES256 or EdDSA), the private key lives only in the auth service and everyone else verifies with the public key, published at a JWKS endpoint:

```
GET https://auth.myapp.com/.well-known/jwks.json
```

Always include the `kid` (key ID) header when signing. That's what enables zero-downtime key rotation: you publish the new key in the JWKS, sign with it, and verifiers pick the right key by its `kid` while old tokens finish expiring.

Where to store tokens in the browser

Not in localStorage. Any XSS —yours or a third-party dependency's— can read it and exfiltrate the tokens in one line. The correct option for web applications is cookies with these attributes:

```
Set-Cookie: refresh_token=8fK2...;
  HttpOnly; Secure; SameSite=Strict;
  Path=/auth/refresh; Max-Age=1209600
```

`HttpOnly` makes it invisible to JavaScript. `Secure` restricts it to HTTPS. `SameSite=Strict` cuts off classic CSRF. And the detail almost everyone skips: `Path=/auth/refresh` means the refresh token only travels to the refresh endpoint, not on every API request. The access token can live in memory (a variable, not storage) and be silently renewed when it expires.

Refresh token rotation with reuse detection

Here's the heart of the guide. Rotation means every refresh token is single-use: when redeemed, the server invalidates it and issues a new one. All tokens descending from the original login form a family.

Reuse detection is the elegant consequence: if someone presents a token that was already redeemed, there are two holders of the same token —the legitimate one and the thief— and you can't tell which is which. The correct response is to revoke the entire family and force a fresh login. A stolen refresh token goes from an indefinitely hijacked session to the minor annoyance of signing in again.

Implementation with FastAPI and SQL (tokens are stored hashed: if your table gets dumped, the attacker gains no usable credentials):

```
import datetime as dt
import hashlib
```

```
import secrets

import jwt  # PyJWT

ACCESS_TTL = dt.timedelta(minutes=10)
REFRESH_TTL = dt.timedelta(days=14)

def _hash(token: str) -> str:
    return hashlib.sha256(token.encode()).hexdigest()

def issue_access_token(user_id: str, session_id: str) -> str:
    now = dt.datetime.now(dt.timezone.utc)
    payload = {
        "iss": "https://api.myapp.com",
        "aud": "myapp-web",
        "sub": user_id,
        "sid": session_id,
        "iat": now,
        "exp": now + ACCESS_TTL,
        "jti": secrets.token_urlsafe(8),
    }
    return jwt.encode(
        payload, PRIVATE_KEY, algorithm="ES256",
        headers={"kid": ACTIVE_KEY_ID},
    )

async def rotate_refresh_token(db, presented: str):
    row = await db.fetchrow(
        "SELECT * FROM refresh_tokens WHERE token_hash = $1",
        _hash(presented),
    )
    if row is None:
        raise AuthError(401, "invalid_token")

    if row["revoked_at"] is not None:
        # Reuse detected: this token was already redeemed.
        # Someone else holds a copy. Burn the whole family.
        await db.execute(
            "UPDATE refresh_tokens SET revoked_at = now() "
            "WHERE family_id = $1 AND revoked_at IS NULL",
            row["family_id"],
        )
        raise AuthError(401, "reuse_detected")

    if row["expires_at"] < dt.datetime.now(dt.timezone.utc):
        raise AuthError(401, "expired_token")
```

```
    # Normal redemption: invalidate the presented one, issue the next
    await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() WHERE id = $1",
        row["id"],
    )
    new_refresh = secrets.token_urlsafe(48)
    await db.execute(
        "INSERT INTO refresh_tokens "
        "(token_hash, family_id, user_id, expires_at) "
        "VALUES ($1, $2, $3, $4)",
        _hash(new_refresh), row["family_id"], row["user_id"],
        dt.datetime.now(dt.timezone.utc) + REFRESH_TTL,
    )
    access = issue_access_token(row["user_id"], str(row["family_id"]))
    return access, new_refresh
```

Two production nuances. First, the whole family needs an absolute expiry (say, 30 days from login): rotation renews tokens, not eternal sessions. Second, allow a few seconds of grace for concurrent redemption — two tabs refreshing at once, or a network retry— before declaring reuse, or you'll generate phantom logouts that cost you hours of support.

Strict access token verification

Most historical JWT CVEs aren't cryptographic flaws: they're permissive verifiers. The rule is to pin, on the server, everything the token claims about itself. In TypeScript with jose:

```
import { createRemoteJWKSet, jwtVerify } from "jose";

const jwks = createRemoteJWKSet(
  new URL("https://auth.myapp.com/.well-known/jwks.json"),
);

export async function verifyAccessToken(token: string) {
  const { payload } = await jwtVerify(token, jwks, {
    issuer: "https://api.myapp.com",
    audience: "myapp-web",
    algorithms: ["ES256"],  // pinned, never from the header
    clockTolerance: 5,      // seconds of clock skew tolerance
  });
  return payload;
}
```

Pinning `algorithms` neutralizes the classic confusion attacks: the infamous `alg: "none"`, and the RS256-to-HS256 downgrade where the attacker signs with the public key used as an HMAC secret.

Requiring `issuer` and `audience` stops you from accepting tokens issued for another context. Clock tolerance should be seconds; if you need minutes, fix your NTP, not your verifier.

Logout and real revocation

With the architecture above, logging out is two operations: revoking the refresh token family (state in your database) and neutralizing the current access token. For the latter, a Redis denylist with a TTL equal to the token's remaining lifetime:

```
async def logout(redis, db, payload, family_id):
    await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() "
        "WHERE family_id = $1 AND revoked_at IS NULL",
        family_id,
    )
    remaining = payload["exp"] - int(time.time())
    if remaining > 0:
        await redis.setex(f"deny:{payload['jti']}", remaining, "1")
```

The middleware checks `deny:<jti>` on every request. Yes, that reintroduces a stateful read, but it's a sub-millisecond Redis lookup over a tiny set (only revoked, unexpired tokens), and you only pay it if you need instant logout. Many systems accept the alternative: without a denylist, a logout leaves the access token alive for a few minutes. Decide explicitly; don't let the tutorial you copied make that decision for you.

The next step: sender-constrained tokens (DPoP)

Rotation detects theft; DPoP (RFC 9449) makes it useless. The client generates a key pair and, on every request, attaches a signed proof (an ephemeral JWT with the HTTP method, URL, timestamp, and a unique `jti`). The authorization server binds tokens to that public key: a stolen token without the private key is worth nothing.

```
POST /auth/token HTTP/1.1
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2Iiwiandr...
```

OAuth 2.1 recommends sender-constrained tokens for public clients, and the AI agent ecosystem (including MCP) pushes in the same direction: agents are exactly the kind of public client with long-lived tokens where sender-constraining pays off most. For a SPA in 2026 it's still optional hardening —rotation with reuse detection covers the common case— but if you're designing a new system with untrusted clients today, evaluate it from the start.

The mistakes that end in an incident

Access tokens lasting hours or days. Defeats the design's primary defense. 5–15 minutes is the standard; silent refresh does the rest.

Refresh tokens stored in plaintext. A table dump becomes stolen sessions. Always hash (SHA-256 is enough: these are high-entropy values, not passwords).

Accepting the algorithm the header declares. The verifier fixes the algorithm list. Always.

Not verifying aud and iss. A token from your staging environment or another service shouldn't open production.

Tokens in localStorage. Every dependency in your bundle is an XSS vector with direct access to credentials.

Logout that only deletes the cookie. If you don't revoke server-side, the session stays alive for whoever holds the tokens.

Rotation without a grace window. Two concurrent tabs trigger false reuse positives and log out legitimate sessions.

Sensitive data in the payload. Base64 is not encryption; any intermediate proxy or log can read it.

Production checklist

JWT access token of 5–15 minutes, signed with ES256/EdDSA and a kid header.

Opaque refresh token, hashed in the database, rotated on every use.

Reuse detection that revokes the whole family, with grace for concurrent redemptions.

Absolute session expiry independent of rotation.

HttpOnly; Secure; SameSite cookies with Path scoped to the refresh endpoint.

Verification with pinned algorithm, iss, and aud, and clock tolerance in seconds.

Published JWKS and signing-key rotation that's been tested, not just documented.

Logout that revokes refresh tokens and, if you need immediate effect, a jti denylist.

Metrics: refreshes per session, detected reuse events, and latency added by the denylist.

None of this is exotic: it's exactly what Auth0, Okta, or any identity provider does under the hood. The difference between "we use JWT" and a defensible authentication system lives in these details, and none of them requires more than one table, a bit of Redis, and strict verifiers.

Are you sure you need JWT? Classic sessions and the BFF pattern

Before going deeper, it's worth asking the uncomfortable question: if your application is a monolith serving its own frontend, a classic cookie session —a random identifier pointing to server-side state— solves the problem with fewer moving parts. Instant revocation, no rotations, no JWKS. JWT wins when there are multiple services verifying identity, APIs consumed by third parties, or an identity provider separate from your APIs. If you have none of those three, the complexity in this guide is optional.

For SPAs there's also a middle ground the IETF now formally recommends: the **BFF (Backend for Frontend)** pattern. The official OAuth draft for browser-based applications (draft-ietf-oauth-browser-based-apps) is explicit: the browser is a hostile environment and, if you can, don't hand it tokens at all. With BFF, a thin backend acts as a confidential client: it keeps access and refresh tokens on the server and gives the

browser only an HttpOnly session cookie. Your SPA talks to the BFF, the BFF attaches the right access token to each request toward your APIs, and there is no token an XSS could exfiltrate.

The cost is an extra network hop and one more component to operate. The practical decision: if your SPA already has its own backend (Next.js, Remix, your own API gateway), the BFF is almost free and it's the most secure option. If your frontend is pure static files against third-party APIs, the browser-token architecture in this guide —HttpOnly cookies for the refresh token, access token in memory— is the reasonable standard.

The complete database schema

The whole rotation system rests on one table. Here's the full definition with the indexes it needs in production:

```
CREATE TABLE refresh_tokens (
  id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  token_hash  text NOT NULL UNIQUE,
  family_id   uuid NOT NULL,
  user_id     uuid NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  created_at  timestamptz NOT NULL DEFAULT now(),
  expires_at  timestamptz NOT NULL,
  revoked_at  timestamptz,
  -- absolute expiry for the family (session)
  family_expires_at timestamptz NOT NULL,
  -- metadata for the "my sessions" view
  user_agent  text,
  ip_hash     text
);

-- The main lookup: redemption by hash
-- (UNIQUE already creates the index on token_hash)

-- Revoking an entire family
CREATE INDEX idx_rt_family
  ON refresh_tokens (family_id)
  WHERE revoked_at IS NULL;

-- Listing a user's active sessions
CREATE INDEX idx_rt_user_active
  ON refresh_tokens (user_id)
  WHERE revoked_at IS NULL;
```

The partial indexes with `WHERE revoked_at IS NULL` keep the structures small: they only index live tokens, which is all you query on the hot path. The `family_expires_at` column implements the absolute session expiry we discussed earlier: it's set at login and never renewed by rotations.

The table grows with every refresh, so it needs cleanup. A periodic job with pg_cron is enough:

```
-- Deletes tokens expired or revoked more than 30 days ago.
-- They're kept for a while for auditing and incident
-- investigation, not deleted immediately.
SELECT cron.schedule(
  'purge-refresh-tokens',
  '15 4 * * *',
  $$
  DELETE FROM refresh_tokens
  WHERE expires_at < now() - interval '30 days'
     OR revoked_at < now() - interval '30 days'
  $$
);
```

Why keep revoked tokens around for a few weeks? Because when you investigate a reuse event you'll want to reconstruct the chain: which token spawned which, from what IP, with what user agent. Delete instantly and you delete the evidence.

Silent refresh on the frontend without race conditions

The backend rotates tokens; the frontend has to consume them without the user noticing. The correct pattern has three pieces: the access token lives in a module variable (not in storage), requests that get a 401 trigger a refresh, and —the part almost everyone gets wrong— concurrent refreshes are deduplicated with single-flight: if five requests fail at once, only one calls the refresh endpoint and the other four await that same promise.

```
// auth-client.ts
let accessToken: string | null = null;
let refreshing: Promise<string> | null = null;

async function refreshAccessToken(): Promise<string> {
  // single-flight: reuse the in-flight refresh
  if (refreshing) return refreshing;

  refreshing = fetch("/auth/refresh", {
    method: "POST",
    credentials: "include",      // sends the HttpOnly cookie
    headers: { "X-Requested-With": "fetch" },
  })
    .then(async (res) => {
      if (!res.ok) throw new Error("refresh_failed");
      const body = await res.json();
      accessToken = body.access_token;
      return accessToken as string;
    })
    .finally(() => { refreshing = null; });
```

```
  return refreshing;
}

export async function apiFetch(path: string, init: RequestInit = {}) {
  const doFetch = (token: string | null) =>
    fetch(path, {
      ...init,
      headers: {
        ...init.headers,
        Authorization: token ? "Bearer " + token : "",
      },
    });

  let res = await doFetch(accessToken);
  if (res.status === 401) {
    try {
      const fresh = await refreshAccessToken();
      res = await doFetch(fresh);       // exactly one retry
    } catch {
      redirectToLogin();
    }
  }
  return res;
}
```

Details that matter. The post-refresh retry is exactly one: if the second attempt also returns 401, the problem isn't the token, and retrying in a loop only hides the error. The `credentials: "include"` is what makes the refresh-token cookie travel; remember that cookie has `Path=/auth/refresh`, so no other request drags it along. And if your app spends long stretches in the background (sleeping tabs), a `visibilitychange` listener that refreshes on wake keeps the user's first action from paying the 401 + refresh latency.

An alternative to reacting to 401s is proactive refresh: decode the access token's `exp` and renew a few seconds before it expires. It works, but it doesn't remove the need for 401 handling (skewed clocks, revocations), so implement it as an optimization, not a replacement.

CSRF and the refresh endpoint: the SameSite nuances

The refresh-token cookie with `SameSite=Strict` cuts off classic CSRF, but it's worth understanding exactly what it cuts and what it doesn't. SameSite governs when the browser attaches the cookie to cross-site requests. With Strict it never travels from another site; with Lax it travels on top-level GET navigations (clicking a link), but not on POSTs or subresources.

For the refresh endpoint, Strict is right: nobody navigates "to" your refresh endpoint, so you lose nothing. But there are two production nuances. First, if your frontend and API live on different domains (app.myapp.com and api.myapp.com are *same-site*; myapp.com and some-third-party-api.com are not), SameSite cookies won't travel and you'll need to either unify domains or move to the BFF pattern. Second,

SameSite is not a substitute for an explicit defense: a compromised subdomain or an old browser can get around it. The cheap defense is requiring a custom header:

```
@app.post("/auth/refresh")
async def refresh(request: Request):
    # Cross-site form submissions cannot attach custom
    # headers without going through CORS.
    if request.headers.get("x-requested-with") != "fetch":
        raise HTTPException(403, "missing_csrf_header")
    ...
```

An HTML form from another origin can't add `X-Requested-With`; only same-origin JavaScript (or an origin your CORS policy explicitly allows) can. Two lines, and CSRF against refresh is closed even where SameSite doesn't reach.

ES256 keys in practice: generation, JWKS, and rotation

The theory says "sign with ES256 and publish a JWKS". The practice starts with generating the key pair:

```
# EC P-256 private key (the ES256 curve)
openssl ecparam -name prime256v1 -genkey -noout \
  -out jwt-signing-2026-08.pem

# Derived public key
openssl ec -in jwt-signing-2026-08.pem -pubout \
  -out jwt-signing-2026-08.pub.pem
```

The dated filename isn't an accident: the key identifier (`kid`) can be that same label. The JWKS endpoint exposes the active public keys in the standard format:

```
from jwcrypto import jwk

@app.get("/.well-known/jwks.json")
async def jwks():
    keys = []
    for kid, pem in ACTIVE_PUBLIC_KEYS.items():
        key = jwk.JWK.from_pem(pem.encode())
        data = key.export_public(as_dict=True)
        data.update({"kid": kid, "use": "sig", "alg": "ES256"})
        keys.append(data)
    # Short Cache-Control: verifiers refresh promptly
    # when you publish a new key
    return JSONResponse(
        {"keys": keys},
```

```
        headers={"Cache-Control": "public, max-age=300"},
    )
```

Zero-downtime rotation is a four-step procedure worth rehearsing, not just documenting:

1. Generate the new pair and publish it in the JWKS *alongside* the old one. Wait at least the JWKS cache TTL (the max-age above plus whatever margin client libraries add).

2. Switch signing: new tokens go out with the new `kid`. Verifiers pick keys by `kid`, so both generations coexist.

3. Wait for the last token signed with the old key to expire (your access token TTL, with margin).

4. Remove the old key from the JWKS and destroy the private key.

With 10-minute access tokens, the whole process fits in half an hour. Schedule it quarterly: a rotation you've only done in theory is a rotation that will fail the day you need it urgently (a compromised key). Libraries like jose in Node cache the JWKS and refresh it on their own when they meet an unknown `kid`; in Python, cache the fetch with a TTL of minutes and invalidate on an unknown `kid`.

Multi-device sessions: the "my sessions" view

Every login creates a refresh token family, and that gives you for free something users expect: the active sessions screen with its "sign out this session" and "sign out everywhere else" buttons. The query is direct thanks to the metadata we store per family:

```
-- Active sessions: the newest live token of each family
SELECT DISTINCT ON (family_id)
       family_id,
       created_at AS last_refresh,
       user_agent,
       ip_hash
FROM refresh_tokens
WHERE user_id = $1
  AND revoked_at IS NULL
  AND expires_at > now()
ORDER BY family_id, created_at DESC;
```

```
@app.delete("/auth/sessions/{family_id}")
async def revoke_session(family_id: str, user=Depends(current_user)):
    result = await db.execute(
        "UPDATE refresh_tokens SET revoked_at = now() "
        "WHERE family_id = $1 AND user_id = $2 "
        "AND revoked_at IS NULL",
        family_id, user.id,
    )
    # That session's access token stays alive for a few
```

```
    # minutes unless you also denylist it by sid.
    return {"revoked": True}
```

Note the filter on `user_id` in addition to `family_id`: without it, any authenticated user could revoke other people's sessions by guessing UUIDs. And the final comment matters: revoking the family cuts future refreshes, but the in-flight access token remains valid until it expires. If your product promises "sign out that other device *now*", add that family's `sid` to the Redis denylist just like on logout.

The special case you must not forget: password change. The universal user expectation is that changing the password kicks out anyone who was inside. Revoke all the user's families except the current session's, and denylist the other families' `sid` values.

Microservices: where to verify and how to propagate identity

With several services behind a gateway there are two models. First: every service verifies the full JWT (signature against JWKS, iss, aud, exp). It's the most robust —no service trusts anything it can't check— and the cost is minimal because the public key is cached and ES256 verification takes microseconds. Second: the gateway verifies once and passes identity inward in an internal header. It saves some CPU, but it turns your internal network into a trust zone: anything that can talk to an internal service can forge the header. It's only acceptable if the internal network has mTLS or mesh policies guaranteeing the header can only come from the gateway.

The practical recommendation is the first: full verification at every edge, and if the gateway already verified, have services repeat it anyway. Cheap defense in depth.

When one service calls another *on behalf of the user*, resist the temptation to forward the original access token as-is. That token has a specific audience (`aud: "myapp-web"`), and if service B accepts it, you're back to the multi-purpose tokens `aud` was meant to prevent. The standard mechanism is **token exchange** (RFC 8693): service A presents the user's token to the authorization server and receives a new token with service B's audience, preserving the original `sub` and recording the delegation chain in the `act` claim. For service-to-service calls with no user involved (scheduled jobs, queues), the right flow is client_credentials with the service's own identity, never a recycled user token.

When Redis goes down: fail-open or fail-closed

The denylist introduces a dependency on the hot path, and dependencies fail. You have to decide in advance what happens to verifications when Redis doesn't answer. **Fail-open** (if Redis doesn't respond, the token is accepted) prioritizes availability: a logout may take a few minutes to become effective during the incident, but your API keeps serving. **Fail-closed** (if Redis doesn't respond, 401 for everyone) prioritizes security and takes the whole platform down with the cache.

For most applications, fail-open with short TTLs is the defensible answer: the risk window is bounded by the access token TTL (minutes) and only affects tokens that were explicitly revoked during the incident. Implement it with an aggressive timeout and metrics:

```
async def is_denylisted(redis, jti: str) -> bool:
    try:
        return await asyncio.wait_for(
```

```
            redis.exists("deny:" + jti), timeout=0.05
        ) == 1
    except (asyncio.TimeoutError, RedisError):
        DENYLIST_CHECK_FAILURES.inc()   # Prometheus metric
        return False   # fail-open: an explicit decision
```

If you handle high-risk operations (transfers, credential changes), apply selective fail-closed only on those endpoints. Granularity is your friend: not all of the API needs the same posture.

Observability and testing for the auth system

A token system without metrics is a black box you only understand during the incident. The minimum you want to export: refreshes by outcome (success, expired, reuse detected), failed verifications by reason (signature, exp, aud), and denylist latency.

```
from prometheus_client import Counter

TOKEN_REFRESH = Counter(
    "auth_token_refresh_total",
    "Refresh token redemptions",
    ["result"],  # success | expired | reuse_detected
)
TOKEN_VERIFY_FAIL = Counter(
    "auth_token_verify_failures_total",
    "Failed access token verifications",
    ["reason"],  # bad_signature | expired | bad_aud | denylisted
)
```

Two alerts are especially worth it. A spike of `reuse_detected` above the noise floor (false positives from concurrent tabs should hover near zero with the grace window) smells like token theft or a client bug; both deserve immediate attention. And a spike of `bad_signature` right after a key rotation means some verifier didn't refresh the JWKS: you have a deployment with a poisoned cache.

As for tests, rotation logic is exactly the kind of code that deserves an exhaustive suite, because its failures are silent until they're catastrophic:

```
async def test_reuse_detection_burns_family(db):
    t1 = await login(db, user_id="u1")          # new family
    t2 = await rotate(db, t1)                   # normal redemption
    with pytest.raises(AuthError) as e:
        await rotate(db, t1)                    # reuse of t1
    assert e.value.code == "reuse_detected"
    # t2 must have died with the family too
    with pytest.raises(AuthError):
        await rotate(db, t2)
```

```
async def test_concurrent_refresh_grace(db):
    t1 = await login(db, user_id="u1")
    # two tabs redeem the same token almost simultaneously
    r1, r2 = await asyncio.gather(
        rotate(db, t1), rotate(db, t1),
        return_exceptions=True,
    )
    # within the grace window: no session dies
    assert not all(isinstance(r, AuthError) for r in (r1, r2))

def test_verifier_rejects_none_alg():
    forged = make_token_with_alg_none(sub="u1")
    with pytest.raises(InvalidTokenError):
        verify_access_token(forged)
```

The third test looks paranoid until you remember that `alg: "none"` was a real CVE in multiple libraries. Verifying that your verifier rejects what it must reject is as important as verifying it accepts what's valid. Add cases for expired tokens with different clock skews, wrong audience, and unknown `kid`, and you'll have the best cost/benefit section of your entire test suite.

Frequently asked questions

Can I use a JWT as the session cookie and skip refresh tokens? You can, but you inherit the worst of both worlds: either the JWT is short-lived and the user gets logged out every 15 minutes, or it's long-lived and you can't revoke it. The access + refresh pair exists precisely so you don't have to choose between those two.

Why not encrypt the payload with JWE instead of just signing? Because it's almost never the real problem. If there's data the client shouldn't read, the simple solution is not putting it in the token: store a reference (`sub`, `sid`) and resolve the rest server-side. JWE adds encryption key management in exchange for hiding something you shouldn't be sending.

Isn't unsalted SHA-256 weak for hashing refresh tokens? Not for this case. Salts and slow algorithms (bcrypt, argon2) exist because human passwords are brute-forceable. A 48-byte random refresh token has ~384 bits of entropy: no dictionary or rainbow table applies. Plain SHA-256 is correct and lets you look up by exact equality with a normal index.

What TTL should the family (absolute expiry) get? It depends on the product's risk. 30 days is reasonable for a general SaaS; banking or sensitive operations go down to hours or a day. The question that decides it: how long is it acceptable for a stolen, unlocked device to keep an open session if the user does nothing?

Access token in a cookie or in the Authorization header? For your own web frontend, an HttpOnly cookie with SameSite for the access token too is defensible (and removes the token from JavaScript's memory entirely). The header wins when your API is consumed by non-browser clients (mobile, CLIs, third parties) and when you want CSRF ruled out by design. Many mature systems end up supporting both.

How often should I rotate signing keys? Absent a compliance requirement saying otherwise, every 90 days is a good balance: enough to bound the damage of an undetected leak, and enough practice that the procedure never rusts.

Worked example: the full flow, from login to logout

The previous pieces —issuance, rotation, cookies, denylist— make sense when you see them assembled. Here's the complete lifecycle of a session in FastAPI, starting with login:

```
@app.post("/auth/login")
async def login(creds: LoginRequest, request: Request, response: Response):
    user = await verify_credentials(creds.email, creds.password)
    if user is None:
        # Same message and same response time whether the
        # email exists or not: don't leak which accounts exist.
        raise HTTPException(401, "invalid_credentials")

    # New family = new session
    family_id = uuid.uuid4()
    refresh = secrets.token_urlsafe(48)
    now = dt.datetime.now(dt.timezone.utc)
    await db.execute(
        "INSERT INTO refresh_tokens "
        "(token_hash, family_id, user_id, expires_at, "
        " family_expires_at, user_agent, ip_hash) "
        "VALUES ($1, $2, $3, $4, $5, $6, $7)",
        _hash(refresh), family_id, user.id,
        now + REFRESH_TTL,
        now + dt.timedelta(days=30),          # absolute expiry
        request.headers.get("user-agent", "")[:300],
        _hash(request.client.host),
    )

    response.set_cookie(
        key="refresh_token",
        value=refresh,
        httponly=True,
        secure=True,
        samesite="strict",
        path="/auth/refresh",
        max_age=int(REFRESH_TTL.total_seconds()),
    )
    access = issue_access_token(str(user.id), str(family_id))
    return {"access_token": access, "expires_in": 600}
```

Three quiet decisions worth commenting on. The response to bad credentials is identical whether the account exists or not, including latency: comparing against a phantom hash when the email doesn't exist keeps the response time from revealing which accounts are registered. The IP is stored hashed: it lets you correlate during an investigation without accumulating personal data in the clear. And the access token comes back in the response body, not in a cookie, because this design keeps it in client memory.

The refresh endpoint ties rotation to the cookie:

```
@app.post("/auth/refresh")
async def refresh_endpoint(request: Request, response: Response):
    if request.headers.get("x-requested-with") != "fetch":
        raise HTTPException(403, "missing_csrf_header")

    presented = request.cookies.get("refresh_token")
    if not presented:
        raise HTTPException(401, "no_refresh_token")

    try:
        access, new_refresh = await rotate_refresh_token(db, presented)
    except AuthError as e:
        # On any failure, clear the cookie: the client will
        # know a full login is due.
        response.delete_cookie("refresh_token", path="/auth/refresh")
        raise HTTPException(401, e.code)

    response.set_cookie(
        key="refresh_token", value=new_refresh,
        httponly=True, secure=True, samesite="strict",
        path="/auth/refresh",
        max_age=int(REFRESH_TTL.total_seconds()),
    )
    return {"access_token": access, "expires_in": 600}
```

And logout revokes server-side before cleaning up client-side:

```
@app.post("/auth/logout")
async def logout_endpoint(request: Request, response: Response,
                          payload=Depends(verify_access)):
    await logout(redis, db, payload, payload["sid"])
    response.delete_cookie("refresh_token", path="/auth/refresh")
    return {"ok": True}
```

With this, the loop is closed: login creates the family and seeds both credentials; each refresh burns one link and forges the next; logout kills the family and neutralizes the current access token. Any token that shows up outside that lane —an already-redeemed refresh, a denylisted access token— is hostile noise the system rejects on its own.

Mobile and native clients: no cookies, same design

On iOS and Android there are no cookies to manage and no XSS to hide from, but the storage problem reappears with a different face: a device can be lost, rooted, or shared. The rule is to keep the refresh token in the system's secure store —Keychain on iOS, Keystore/EncryptedSharedPreferences on Android— and never in plain preferences, files, or local databases.

```
// Android (Kotlin): EncryptedSharedPreferences
val masterKey = MasterKey.Builder(context)
    .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
    .build()

val securePrefs = EncryptedSharedPreferences.create(
    context, "auth_store", masterKey,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
securePrefs.edit()
    .putString("refresh_token", newRefreshToken)
    .apply()
```

The rest of the design carries over intact: the access token lives in process memory, the refresh token rotates on every use, and reuse detection works exactly the same. The mobile nuances are operational. First, background refresh: if the OS suspends your app for days, the refresh token may expire; handle it with a soft login (biometrics against the Keychain) instead of dumping the user at the password form. Second, mobile's aggressive network retries make concurrent redemption more likely: the server's grace window stops being optional. Third, if your app has widgets or extensions, every extra process is another consumer of the same refresh token; consider separate families per surface so you can revoke them independently.

Rate limiting the authentication endpoints

Auth endpoints are the most attacked part of any API: login gets credential stuffing (leaked password lists tried at scale) and refresh gets brute force against tokens. Rate limiting here isn't your API's generic limiter; it needs its own dimensions.

For login, limit on two keys at once: per IP (stops mass scanning from one origin) and per target account (stops a distributed attack against one user). Reasonable thresholds differ: a legitimate IP behind a corporate NAT can produce dozens of logins per minute; an individual account cannot.

```
async def check_login_limits(redis, email: str, ip: str):
    # Per IP: 30 attempts / 5 min (NAT-friendly)
    ip_key = "rl:login:ip:" + _hash(ip)
    # Per account: 5 attempts / 5 min
    acct_key = "rl:login:acct:" + _hash(email.lower())

    async with redis.pipeline() as pipe:
```

```
        pipe.incr(ip_key);  pipe.expire(ip_key, 300, nx=True)
        pipe.incr(acct_key); pipe.expire(acct_key, 300, nx=True)
        ip_n, _, acct_n, _ = await pipe.execute()

    if ip_n > 30 or acct_n > 5:
        # 429 with Retry-After, always the same body
        raise HTTPException(
            429, "too_many_attempts",
            headers={"Retry-After": "300"},
        )
```

The per-account limit also turns blocking into signal: if an account keeps hitting its limit without a successful login, notify the owner ("we've detected access attempts") and consider requiring a second factor on the next successful login. For the refresh endpoint the pattern is different: a healthy client refreshes at most a few times per minute per session, so a per-family limit (not per IP) of, say, 10 per minute catches broken clients and attacks alike without punishing anyone legitimate.

Roles and permissions inside the token: freshness versus latency

Putting roles in the JWT (`"role": "admin"`) is convenient: authorization resolves without touching the database. But claims stay frozen until the token expires, and that turns every authorization decision into a question about freshness: if you demote a malicious admin, how many minutes of admin do they have left? With 10-minute access tokens, the answer is "up to 10", and for most products that's acceptable. For destructive operations it isn't.

The mature pattern is two-tiered. Cheap, frequent checks (can they see this page? can they call this read endpoint?) use the token's claims. Sensitive operations (delete the project, change other people's permissions, export data) revalidate against the database at that moment, ignoring whatever the token says. You pay the query only where freshness matters.

Watch the size too. Every claim travels on every request, and intermediate proxies commonly cap headers at 8 KB; a token carrying long permission lists or group memberships gets dangerously close. If your permission model doesn't fit in a handful of claims, the token should carry the reference (`sub`, `sid`, maybe a coarse `role`) and the detail should live server-side, cached if needed. A JWT is not a portable database.

Anatomy of an incident: how all this works under fire

To ground the design's value, a plausible incident from start to finish. Monday, 10:40: the `reuse_detected` alert climbs from its noise floor (zero or one a day) to fourteen events in an hour. It's not a new deployment and not a known client bug.

10:55: with the token table and its metadata, the team reconstructs the affected chains. The fourteen families share a pattern: the legitimate redemption came from each user's usual IPs and user agents; the second redemption —the one that tripped detection— came from a single IP range at a cheap hosting provider. Provisional conclusion: someone is exfiltrating refresh tokens from a subset of users and redeeming them from their own infrastructure.

11:10: containment. The fourteen families already revoked themselves (that's reuse detection working); the team additionally revokes all families for those users, pushes their `sid` values into the denylist, and forces re-login. Damage per user is bounded to the remaining minutes of the last stolen access token.

11:30: how did the tokens leak? The affected users share an app version and had all installed one specific browser extension. It's not an XSS on the site: it's a malicious extension reading the page's storage. And here comes the design's dividend: because the refresh token lives in an HttpOnly cookie with a scoped Path and the access token in memory, the extension only captured access tokens with minutes to live. Without rotation and properly set cookies, that same incident would have been "sessions stolen indefinitely until someone notices".

12:00: wrap-up. Signing keys rotated as a precaution (the rehearsed, zero-downtime procedure from the earlier section), affected users notified, and one new alert rule: refresh redemptions from hosting ASNs when the family was born on a residential IP. The post-mortem doesn't propose redesigning anything: it proposes lowering the access token TTL from 10 to 5 minutes. The system worked.

Migrations: getting to this design from where you are

You rarely build this from scratch; usually you migrate from something. Two common routes.

From HS256 to ES256 without logging anyone out. The trick is that verifying and issuing are independent operations. Step one: verifiers accept both algorithms, each with its own key, choosing by `kid` (old HS256 tokens carry no `kid`, which already identifies them). Step two: the issuer switches to ES256. Step three: once the last HS256 token has expired —one access-token TTL later— HS256 support is removed from the verifier. Total: three deployments and zero users logged out.

```
def verify_transitional(token: str):
    header = jwt.get_unverified_header(token)
    if "kid" in header:                    # new generation
        return jwt.decode(
            token, public_key_for(header["kid"]),
            algorithms=["ES256"],
            audience="myapp-web", issuer=ISS,
        )
    # old generation, being phased out
    LEGACY_HS256_VERIFIED.inc()            # metric: must trend to 0
    return jwt.decode(
        token, LEGACY_SECRET, algorithms=["HS256"],
        audience="myapp-web", issuer=ISS,
    )
```

The legacy-path metric is the operational key: once it's been flat at zero for a few days, you can delete the branch with confidence instead of faith.

From classic sessions to access + refresh. Don't migrate users in bulk: let the migration happen on its own. The middleware temporarily accepts both credentials (old session cookie or new Bearer). New logins issue the token pair; old sessions keep working until they expire naturally. Within a few weeks, classic-

session traffic dries up and you remove the code. The alternative —invalidating every session one Tuesday and forcing a global login— is technically simpler and operationally worse: the support spikes aren't worth it.

The library bugs worth knowing by name

JWT's history is full of vulnerabilities that weren't in the standard but in specific implementations, and knowing them vaccinates you against their patterns.

The classic `alg: "none"` (CVE-2015-9235 in node-jsonwebtoken, among others) allowed presenting unsigned tokens that some libraries happily accepted. The permanent lesson: the verifier pins the algorithm list, never reads it from the token. The RS256/HS256 confusion —signing with the public key used as an HMAC secret— resurfaced for years across different libraries; in 2024 it was python-jose's turn (CVE-2024-33663, algorithm confusion with ECDSA keys). And the eternal application-level mistake: calling the decode function without verification —`jwt.decode` with verification disabled, or jsonwebtoken's `decode()` which verifies nothing, versus `verify()` which does— and using those claims for authorization. Audit your code for those calls: it's a one-minute grep that has found holes at more than one company.

In today's Python ecosystem, PyJWT is the most actively maintained option, with an API that forces you to pass `algorithms` explicitly. In Node, jose is the modern reference (remote JWKS, EdDSA, and DPoP support) next to the veteran jsonwebtoken. Whichever you choose, pin the version, subscribe to its security advisories, and treat any verifier update as a sensitive change that goes through review.

Quick glossary

Access token. Short-lived credential (minutes) that authorizes API requests; here, a statelessly verifiable JWT.

Refresh token. Long-lived opaque credential whose only power is obtaining new access tokens; lives hashed on the server.

Family. The full lineage of refresh tokens descending from one login; the unit of revocation for a session.

Rotation. The policy making each refresh token single-use, with each redemption issuing the next one.

Reuse detection. The alarm that fires when an already-redeemed token is presented; it implies two holders and burns the family.

JWKS. JSON document with the public verification keys, published at a well-known URL.

kid. Key identifier in the JWT header; lets multiple keys coexist and rotate without downtime.

Denylist. List of tokens revoked before their expiry, checked on every verification; in this design, a small TTL'd set in Redis.

DPoP. Mechanism (RFC 9449) that binds tokens to a client-held key, making a stolen token useless without that key.

BFF. Backend for Frontend: a thin backend that keeps tokens server-side and hands the browser only a session cookie.

Single-use tokens: password reset, magic links, and email change

Orbiting the session system are other tokens that look similar and have opposite requirements: the "reset your password" link, the passwordless magic link, the email-change confirmation. The temptation is to solve them with signed JWTs —"I already have the infrastructure..."— and it's exactly the wrong tool. These tokens must be single-use, and a signed JWT can't be single-use without server-side state: signed once, it's valid until it expires, whether it's redeemed once or twenty times.

The correct shape is the same as refresh tokens: opaque random value, hashed in the database, short expiry, and a column recording redemption.

```
CREATE TABLE one_time_tokens (
  token_hash  text PRIMARY KEY,
  purpose     text NOT NULL,   -- password_reset | magic_link | email_change
  user_id     uuid NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  payload     jsonb,           -- e.g. the proposed new email
  expires_at  timestamptz NOT NULL,
  used_at     timestamptz
);

-- Atomic redemption: only one request wins even if two arrive
UPDATE one_time_tokens
SET used_at = now()
WHERE token_hash = $1
  AND purpose = $2
  AND used_at IS NULL
  AND expires_at > now()
RETURNING user_id, payload;
```

The conditional UPDATE with RETURNING is the detail that prevents the race: if two requests redeem the same link simultaneously, only one gets a row back; the other gets zero rows and a clean error. Expirations should be genuinely short: 15-30 minutes for a password reset, 5-10 for a magic link. And redeeming a password reset must ripple into the session system we already built: a new password means revoking all the user's families and emptying their sessions, because the most common reason for a reset is precisely suspected compromise.

Email change deserves its own choreography, because it's the favorite target of anyone who hijacks a session (by changing the email, the attacker inherits all future resets). The robust pattern is two-phase: the confirmation link goes to the *new* email (proof of possession), but the notice with a cancellation option goes to the *old* email, carrying its own single-use token that reverts the change and revokes sessions. That reversal window is the difference between a scare and a lost account.

The real cost: latency numbers to decide with data

Every decision in this guide (stateless or denylist? ES256 or HS256? verify in every service?) is at bottom a latency-budget decision, so let's close with the orders of magnitude that govern them. Verifying an HS256 signature costs around a microsecond; ES256, tens of microseconds (and signing ES256 is cheaper than

verifying it). A Redis lookup on the same network, half a millisecond; an indexed PostgreSQL query, one to two milliseconds under normal load.

Read them together and the conclusions fall out on their own. The difference between HS256 and ES256 is irrelevant next to any I/O: choosing an algorithm for performance is optimizing 0.1% of the budget; choose for operational security (the private key in one place), which is where the real difference lives. Verifying the JWT in every microservice adds microseconds per hop: defense in depth is free for practical purposes. The Redis denylist adds ~0.5 ms per request: noticeable in a tight p99, acceptable almost always, and with the fail-open policy Redis's bad day doesn't become your API's bad day. And the classic-sessions alternative —one query per request— costs one or two milliseconds plus a hard database dependency on every endpoint: perfectly valid for a monolith, expensive for twenty services.

The whole guide's moral fits in one sentence: you don't pick tokens by fashion but by budget —latency, operations, and risk— and a well-built access + refresh system is how you pay little of all three at once.