Federated Identity Integration with Legacy Database Authentication: Patterns for Secure Identity Propagation in Hybrid Environments

Pablo Andrés González Casco[1,*1][0009-0001-1038-0428], Sebastián Báez Escobar[1,*][0009-0001-0236-9384], Gabriel Nahuel Milanesi[1][0009-0002-5276-6860], Cristina Isabel Solan[1][0009-0000-6139-8719], Ezequiel Merino Spinedi[1][0009-0009-0194-9481], Elizabeth Minchaca[1][0009-0008-8357-5956], and Ana Mariela Cossio Aquino[1][0009-0004-0604-8475]

[1] *Fiscalía de Estado de la Provincia de Buenos Aires, Buenos Aires, Argentina*

pablogon@gmail.com · sebaez14@gmail.com · nahuelkub@gmail.com · solan.cristina@gmail.com · merino.eze@gmail.com · elizabeth.minchaca@gmail.com · cossio.ana.mariela@gmail.com

Abstract. Legacy Windows-based enterprise systems typically enforced security at the database layer, granting domain users direct access to SQL Server and relying on stored procedures to constrain permissible operations. As organizations adopt modern web APIs and OpenID Connect (OIDC)-based single sign-on, a structural mismatch emerges: SQL Server cannot authenticate JSON Web Tokens (JWT) natively, forcing applications to use technical service accounts for database connections while the real user identity is known only at the application layer. This gap is particularly acute in hybrid environments where legacy desktop applications continue to require direct database connectivity alongside modern API-driven clients, making it impossible to enforce network-level access restrictions uniformly. This paper characterizes this security problem, analyzes its evolution through different architectural stages, and proposes a classification of patterns for securely propagating user identity from the OIDC/JWT layer down to SQL Server. Three connection models and three identity propagation mechanisms are evaluated: parameter passing, EXECUTE AS impersonation, and SESSION_CONTEXT injection. Each combination is assessed across security guarantees, operational complexity, and compatibility with legacy constraints. While EXECUTE AS impersonation provides native engine-level audit (SQL Server records the real user identity in audit specifications, triggers, and traces), it requires each user to have a login in SQL Server, either individually or via a domain group. The SESSION_CONTEXT approach with connection interceptors emerges as the recommended pattern for progressive migration scenarios: it enables application-layer user traceability without individual database logins or mass stored procedure refactoring, at the cost of delegating audit persistence to the application layer.

---

[*] *Both authors contributed equally to this work. / Ambos autores contribuyeron igualmente a este trabajo.*

Keywords: OpenID Connect (OIDC), federated identity, identity propagation, database authentication, SQL Server security

Integración de identidad federada con autenticación de base de datos legacy: patrones para la propagación segura de identidad en entornos híbridos

Resumen. Los sistemas empresariales legacy basados en entornos Windows implementaban la seguridad a nivel de base de datos, otorgando a los usuarios de dominio acceso directo a SQL Server y usando stored procedures para restringir las operaciones permitidas. Con la adopción de APIs web modernas y el inicio de sesión único basado en OpenID Connect (OIDC), surge una brecha estructural: SQL Server no puede autenticar tokens JWT de forma nativa, lo que obliga a las aplicaciones a usar cuentas técnicas de servicio para la conexión a la base de datos, mientras que la identidad real del usuario queda restringida a la capa de aplicación. Esta brecha es especialmente crítica en entornos híbridos donde aplicaciones de escritorio legacy aún requieren conectividad directa a la base de datos junto a clientes modernos basados en APIs, impidiendo aplicar restricciones de acceso a nivel de red de forma uniforme. Este trabajo caracteriza este problema de seguridad, analiza su evolución a través de distintas etapas arquitectónicas y propone una clasificación de patrones para propagar de forma segura la identidad del usuario desde la capa OIDC/JWT hasta SQL Server. Se evalúan tres modelos de conexión y tres mecanismos de propagación de identidad: parámetro explícito, impersonación con EXECUTE AS e inyección de SESSION_CONTEXT. Cada combinación se analiza en términos de garantías de seguridad, complejidad operativa y compatibilidad con restricciones legacy. La impersonación con EXECUTE AS ofrece auditoría nativa a nivel de motor (SQL Server registra al usuario real en audit specifications, triggers y traces), pero requiere que cada usuario tenga un login en SQL Server, ya sea individual o a través de un grupo de dominio. El enfoque SESSION_CONTEXT con interceptores de conexión emerge como el patrón recomendado para escenarios de migración progresiva: permite la trazabilidad de operaciones por usuario a nivel de aplicación sin requerir logins individuales en la base de datos ni una refactorización masiva de stored procedures, a costa de delegar la persistencia de la auditoría a la capa de aplicación.

Palabras clave: OpenID Connect (OIDC), identidad federada, propagación de identidad, autenticación en bases de datos, seguridad en SQL Server

1 Introducción

La gestión de identidad en sistemas empresariales ha experimentado una transformación profunda en la última década. Los sistemas legacy fueron diseñados bajo un modelo donde los usuarios finales se autenticaban directamente contra la base de datos usando credenciales de dominio (Active Directory), y la seguridad de las operaciones se delegaba al motor de base de datos mediante stored procedures y permisos granulares a nivel de objeto (Bertino & Sandhu, 2005). Este modelo, aunque efectivo en entornos homogéneos, presenta limitaciones estructurales que se vuelven críticas al coexistir con arquitecturas modernas.

La adopción de estándares como OAuth 2.0 (RFC 6749) y OpenID Connect (OIDC Core 1.0) introdujo un paradigma radicalmente diferente: la identidad del usuario se representa mediante tokens firmados digitalmente (JWT, RFC 7519) que se validan en la capa de aplicación, y el acceso a recursos se delega a través de flujos de autorización estandarizados. Sin embargo, esta evolución no elimina inmediatamente la dependencia de las bases de datos relacionales existentes, las cuales no soportan autenticación basada en JWT.

Cuando una organización inicia la transición hacia OIDC, enfrenta invariablemente un período híbrido donde conviven aplicaciones legacy con acceso directo a la base de datos y APIs modernas que validan tokens. En este período, surgen brechas de seguridad difíciles de mitigar sin un modelo claro de propagación de identidad.

Para abordar este problema, el trabajo parte de la caracterización de la brecha de identidad en escenarios de migración progresiva y analiza las vulnerabilidades concretas que introduce la convivencia de modelos. A partir de ese diagnóstico, se propone una clasificación de tres patrones de conexión al motor y tres mecanismos de propagación de identidad, evaluando cada combinación según sus garantías de seguridad, complejidad operativa y compatibilidad con restricciones legacy.

2 El Modelo de Seguridad Legacy

2.1 Arquitectura original

En los sistemas empresariales desarrollados antes de la adopción masiva de aplicaciones web, el modelo de seguridad típico en entornos Microsoft se estructuraba en torno a la identidad de dominio. Los usuarios finales poseían cuentas en Active Directory (AD) del dominio corporativo, y SQL Server se configuraba con Windows Authentication, lo que permitía que las credenciales de dominio se usaran directamente para autenticar las conexiones. La aplicación cliente (generalmente una aplicación de escritorio) se conectaba a la base de datos usando la identidad del usuario logueado en el sistema operativo (Integrated Security / Trusted Connection). La seguridad de las operaciones se implementaba mediante stored procedures: los usuarios recibían únicamente permisos de ejecución sobre estos procedimientos, sin permisos DML directos sobre las tablas.

Este modelo ofrecía una propiedad de seguridad valiosa: la identidad del usuario se obtenía del contexto de seguridad de la conexión mediante funciones como

`SUSER_SNAME()` o `SUSER_SID()`, y no podía ser manipulada desde la aplicación. De este modo, se eliminaba la posibilidad de suplantación de identidad a nivel de aplicación.

2.2 Fortalezas del modelo

El modelo legacy presentaba las siguientes garantías de seguridad:
Control de operaciones: Al restringir los permisos a la ejecución de stored procedures específicos, la base de datos se convertía en la capa de control principal. Un usuario no podía ejecutar operaciones fuera de las contempladas por los procedimientos existentes, independientemente de la herramienta cliente utilizada.
Auditoría nativa: SQL Server registraba las conexiones y operaciones con la identidad real del usuario de dominio. Cualquier actividad quedaba trazable a un individuo específico sin dependencia de logs de aplicación.
Inmutabilidad de la identidad: La identidad no se transmitía como dato en la consulta, sino que surgía del protocolo de autenticación de la conexión. Esto eliminaba una clase entera de vulnerabilidades asociadas al parámetro de usuario.

2.3 Debilidades del modelo

No obstante, el modelo presentaba vulnerabilidades estructurales:
Acceso directo a la base de datos: Un usuario con credenciales válidas podía conectarse a SQL Server con cualquier cliente SQL (SQL Server Management Studio, scripts, herramientas de terceros) y ejecutar stored procedures fuera del contexto de la aplicación. Los permisos de ejecución sobre procedimientos no impedían el uso de herramientas externas.
Sesiones persistentes sin expiración: No existía un mecanismo de control centralizado de sesiones. Una sesión abierta no expiraba por inactividad a menos que se configurara explícitamente, y no podía invalidarse de forma centralizada.
Dependencia de infraestructura de dominio: El modelo requería que todos los clientes estuvieran unidos al dominio y que el SQL Server participara en la infraestructura Kerberos/NTLM del directorio activo, limitando la portabilidad y la containerización.

3 Evolución Hacia Arquitecturas Modernas y Problemas Emergentes

3.1 Primera transición: aplicaciones web con identidad de dominio

Con la adopción inicial de aplicaciones web, muchas organizaciones intentaron mantener el esquema legacy. Las aplicaciones web se configuraban para impersonar al usuario de dominio (mediante Kerberos Constrained Delegation o configuraciones similares) y conectarse a SQL Server con la identidad original del usuario.
Este enfoque amplificó las vulnerabilidades existentes. En el modelo puramente legacy, la aplicación de escritorio era el único vector de acceso a los stored procedures.

En el modelo web con impersonación, cualquier vulnerabilidad en la aplicación (endpoints inseguros, validaciones incorrectas, inyección) podía resultar en la ejecución de operaciones con los privilegios reales del usuario, desde una superficie de ataque radicalmente mayor.
Cuando en esta misma etapa los equipos reemplazan los stored procedures por consultas LINQ u ORM, la restricción a nivel de base de datos desaparece por completo: el usuario impersonado puede ejecutar operaciones arbitrarias sobre las tablas, sin el control que antes ofrecían los procedimientos. Esto elimina la única barrera efectiva del modelo legacy sin sustituirla por ninguna equivalente.

3.2 Segunda transición: APIs y ORMs con cuenta técnica

A medida que las organizaciones adoptaron arquitecturas basadas en APIs REST, la conexión a SQL Server migró hacia cuentas de servicio técnicas (un usuario de AD o un login local de SQL Server), compartidas por todas las instancias de la aplicación. La identidad del usuario pasó a gestionarse exclusivamente a nivel de aplicación.
Esta transición introdujo el problema central de este trabajo: la desconexión entre la identidad autenticada a nivel de aplicación y la identidad visible a nivel de base de datos. El motor de base de datos pierde la capacidad de distinguir qué usuario final originó una operación.
Adicionalmente, al migrar la lógica de restricción de operaciones desde los stored procedures hacia el código de la API (mediante ORMs, LINQ o consultas parametrizadas), la base de datos dejó de ser la capa de control principal. Los riesgos de inyección SQL no desaparecieron (Halfond et al., 2006), sino que su materialización pasó a depender de las prácticas individuales del equipo de desarrollo.

3.3 El problema de los entornos híbridos

La situación más compleja se presenta en organizaciones en transición activa, donde coexisten aplicaciones de escritorio legacy (que requieren conectividad directa a SQL Server con credenciales de dominio) y APIs modernas (que se conectan mediante cuentas técnicas). Este escenario genera una limitación crítica de seguridad: **no es posible restringir el acceso a la base de datos por dirección IP o por origen de conexión**.
Las aplicaciones de escritorio legacy requieren que los puestos de trabajo de los usuarios puedan abrir conexiones TCP al puerto SQL Server (1433 por defecto). Esto implica que cualquier usuario del dominio con permisos válidos puede conectarse a la base de datos desde su estación de trabajo usando herramientas externas, ejecutando los stored procedures disponibles o, si los permisos no están correctamente restringidos, operaciones DML directas.
En este escenario, el modelo híbrido no mantiene la seguridad del modelo legacy ni alcanza la seguridad del modelo moderno. Las debilidades de ambos se acumulan.

3.4 Limitaciones adicionales del modelo legacy en transición

El escenario híbrido expone otras limitaciones. Por un lado, la integración entre servicios frecuentemente usa autenticación básica (usuario/contraseña) para invocar APIs, lo que introduce riesgos de transmisión y almacenamiento de credenciales sin los mecanismos modernos de expiración y revocación que provee OAuth 2.0. Por otro, la dependencia de Windows Authentication impide adoptar contenerización o arquitecturas de microservicios. Finalmente, resulta difícil implementar Multi-Factor Authentication de forma uniforme cuando parte de los accesos ocurre directamente a nivel de motor de base de datos.

4 Patrones para la Integración de Identidad OIDC con SQL Server

4.1 Contexto del problema

En una arquitectura basada en OIDC, el Authorization Code Flow (RFC 6749) se extiende para incluir autenticación: el Identity Provider emite tanto un Access Token como un ID Token con los claims del usuario (OIDC Core 1.0). El flujo completo involucra redirects, emisión de un código de autorización y su intercambio en el token endpoint; a los efectos de este trabajo, se resume en los pasos relevantes para la propagación de identidad hacia la base de datos:

1. El usuario se autentica contra el Identity Provider (IdP).
2. El IdP emite simultáneamente un ID Token (con los claims del usuario: sub, email, preferred_username, etc.) y un Access Token; la aplicación cliente recibe ambos.
3. La API valida la firma del Access Token y extrae los claims.
4. La API autoriza la operación y accede a SQL Server.

El problema radica en el paso 4: SQL Server no tiene capacidad nativa para validar JWTs. A diferencia de servicios cloud como Azure SQL (que soporta Azure AD tokens mediante MSAL), SQL Server on-premises requiere un mecanismo de autenticación tradicional para la conexión. Por lo tanto, la API debe conectarse con una cuenta de servicio técnica, perdiendo la identidad del usuario original a nivel de motor.
La pregunta de diseño es: **¿cómo propagar la identidad OIDC del usuario hasta SQL Server de forma segura y auditable?**
Para responderla, se distinguen dos dimensiones de decisión: (a) el modelo de conexión al motor, y (b) el mecanismo de propagación de identidad dentro de la sesión.

4.2 Alternativas de Conexión al Motor

Alternativa de Conexión 1 (AC1): Cuenta de Servicio de Active Directory

La aplicación se conecta a SQL Server usando un usuario de dominio dedicado (p. ej., `DOMAIN\svc_app`), configurado en el Application Pool de IIS o en las variables de entorno del servidor. La conexión usa Integrated Security (Trusted Connection).

```
Connection string: Server=SQL01; Database=AppDB; Trusted_Connec-
tion=True; App=MyAPI
```

Requisitos: El servidor de aplicaciones debe estar unido al dominio; SQL Server debe pertenecer al mismo dominio o tener confianza.

Análisis de seguridad: Las queries se ejecutan bajo la identidad svc_app, no del usuario real. No hay auditoría nativa por usuario final. El beneficio principal es que no hay contraseña almacenada en la aplicación, las credenciales de la cuenta de servicio se gestionan mediante la infraestructura Kerberos del dominio.

Alternativa de Conexión 2 (AC2): Login Local de SQL Server

La aplicación se conecta usando un login propio de SQL Server (autenticación SQL). La contraseña debe gestionarse mediante variables de entorno o un secret manager (p. ej., HashiCorp Vault, Azure Key Vault).

```
Connection string: Server=SQL01; Database=AppDB; User
Id=svc_app; Password=<secret>
```

Análisis de seguridad: Mismo resultado que AC1 en términos de identidad: todas las queries son visibles bajo svc_app. La ventaja operacional es que no requiere infraestructura de dominio, lo que habilita la containerización. El riesgo adicional es la gestión del secreto de la contraseña.

Alternativa de Conexión 3 (AC3): Login por Usuario SSO

Se crea un login de SQL Server para cada usuario autenticado por el IdP. Un proceso de sincronización mantiene los logins en el motor actualizados con los usuarios del IdP.

Análisis de seguridad: Permite auditoría por usuario real y permisos granulares. Sin embargo, la complejidad operativa es significativa: gestión de contraseñas por usuario, sincronización con el ciclo de vida de usuarios en el IdP (altas, bajas, cambios de rol).

Una simplificación posible para la gestión de contraseñas es asignar a todos los logins una contraseña única compartida, gestionada centralmente como si fuera una cuenta técnica. Esto reduce la complejidad operativa sin sacrificar la trazabilidad, ya que las trazas a nivel motor se basan en el nombre de login, no en la contraseña. El perfil de riesgo de la contraseña es equivalente al de AC1 y AC2: una credencial centralizada que debe protegerse con las mismas prácticas (secret manager, rotación periódica).

4.3 Alternativas de Propagación de Identidad

Las alternativas AC1 y AC2 establecen una conexión técnica compartida. Las alternativas de propagación de identidad son técnicas para hacer disponible la identidad real del usuario dentro de la sesión SQL, una vez establecida la conexión técnica.

Alternativa de Propagación 1 (AP1): Parámetro

La API extrae el claim preferred_username (o equivalente) del JWT validado y lo pasa como parámetro a cada stored procedure.

```
EXEC dbo.RegistrarMovimiento @monto = 1500, @userName = 'domain\jsmith'

SELECT * FROM dbo.Movimientos WHERE userId = (SELECT id FROM dbo.usu-
arios WHERE username = @userName)
```

Análisis de seguridad: La identidad fluye como dato en la consulta, lo que la hace dependiente de la correcta validación del token en la API. Si la API tiene vulnerabilidades, un atacante podría suplantar la identidad pasando un @userName arbitrario.

Impacto en el código base: Requiere modificar todos los stored procedures existentes para aceptar y utilizar el parámetro de usuario, lo que representa un costo de refactorización alto en aplicaciones legacy.

Alternativa de Propagación 2 (AP2): Impersonación con EXECUTE AS

SQL Server permite cambiar el contexto de seguridad de la sesión mediante la instrucción `EXECUTE AS USER` (Microsoft, 2024b). La API, al abrir la conexión, ejecuta automáticamente esta instrucción antes de cualquier operación.

```
EXECUTE AS USER = 'domain\jsmith';
-- operaciones subsiguientes se ejecutan como domain\jsmith
```

Puede implementarse mediante un interceptor de conexión que lo ejecute automáticamente al abrir cada conexión.

Requisitos: El usuario debe tener acceso como login en SQL Server: puede ser un login individual de AD, un login local de SQL Server (AC3), o ser miembro de un grupo de dominio con login en el motor. En los casos de login individual de AD o grupo de dominio, el servidor SQL debe estar unido al dominio. El login de la cuenta técnica debe tener el permiso `IMPERSONATE` sobre los usuarios objetivo.

Análisis de seguridad: Las queries se ejecutan con los permisos efectivos del usuario impersonado, lo que provee **auditoría nativa a nivel de motor**: SQL Server registra al usuario impersonado (no la cuenta técnica) permitiendo construir trazas de auditoría completas por usuario real. Esta es la única alternativa que no requiere ninguna implementación explícita para garantizar trazabilidad completa: el motor la provee nativamente sin cambios en la capa de aplicación ni en los stored procedures. Sin embargo, si el usuario también tiene acceso directo a la base (escenario híbrido), el patrón no cierra la brecha: el usuario puede seguir ejecutando operaciones fuera de la API. Este patrón es más adecuado para arquitecturas completamente modernas donde el acceso directo ya ha sido eliminado.

Debe tenerse en cuenta, además, que el login de SQL Server requerido por AP2 no se limita a habilitar la impersonación: otorga al usuario la capacidad de conectarse directamente a la base de datos con sus propias credenciales. En arquitecturas modernas

donde la cuenta técnica tiene permisos DML sobre las tablas (en lugar de solo EXECUTE sobre stored procedures), un usuario con login puede acceder directamente a esas tablas sin pasar por la API, eludiendo cualquier control de autorización implementado en la capa de aplicación. Por esta razón, AP2 solo es seguro cuando el acceso directo al motor está efectivamente bloqueado a nivel de red.

Alternativa de Propagación 3 (AP3): SESSION_CONTEXT (Recomendado para Migración Progresiva)

SQL Server provee el mecanismo `sp_set_session_context` para almacenar pares clave-valor en el contexto de la sesión actual, con visibilidad restringida a esa sesión (Microsoft, 2024c).
La API establece el contexto de usuario al abrir cada conexión, idealmente mediante un interceptor transparente:

```
EXEC sp_set_session_context N'userName', N'domain\jsmith',
@read_only = 1;
```

El flag `@read_only = 1` es crítico: impide que código posterior en la misma sesión sobreescriba el valor, reduciendo el riesgo de manipulación.

Cualquier consulta en la sesión (stored procedures, consultas ad-hoc o queries generadas por un ORM) puede leer el contexto mediante `SESSION_CONTEXT()` (Microsoft, 2024a):

```
DECLARE @userName NVARCHAR(256) = CAST(SESSION_CONTEXT(N'userName') AS
NVARCHAR(256));
DECLARE @userId INT = (SELECT id FROM dbo.usuarios WHERE username =
@userName AND fechaBorrado IS NULL);
```

Análisis de seguridad: Cuando la API es el único vector de conexión, el `SESSION_CONTEXT` solo puede provenir del interceptor de la API. En escenarios híbridos, en cambio, un usuario de dominio conectado directamente puede forzar un valor con `sp_set_session_context` en su propia sesión; si un stored procedure usara ese valor para identificar al usuario y decidir qué datos puede ver o modificar, podría suplantar a otro, ya que tener solo `EXECUTE` no impide la suplantación dentro de lo que el procedimiento permite. Este riesgo solo aparece cuando un mismo stored procedure compartido resuelve la identidad de las dos formas (por `SUSER_SNAME()` para la aplicación legacy y por `SESSION_CONTEXT` para la nueva), ya que el usuario de dominio tiene `EXECUTE` sobre él y puede forzar el contexto; si cada aplicación usara procedimientos propios, ningún procedimiento-objetivo consultaría el valor forzado. La defensa es centralizar la resolución de identidad en una única función que, según el usuario que estableció la conexión, confía en el `SESSION_CONTEXT` solo para la cuenta técnica de SQL y, para una conexión de dominio, lo ignora y usa el login real `SUSER_SNAME()`; así un valor forzado nunca se consulta. El parámetro `@read_only = 1` protege la sesión de la API frente a inyección, pero no es lo que cubre la conexión directa.

Limitación de auditoría: A diferencia de AP2, SQL Server no registra la identidad del usuario real en sus mecanismos nativos. Las trazas de auditoría muestran únicamente la cuenta técnica. La trazabilidad por usuario queda en la capa de aplicación y su cobertura depende de que esta la implemente explícitamente.

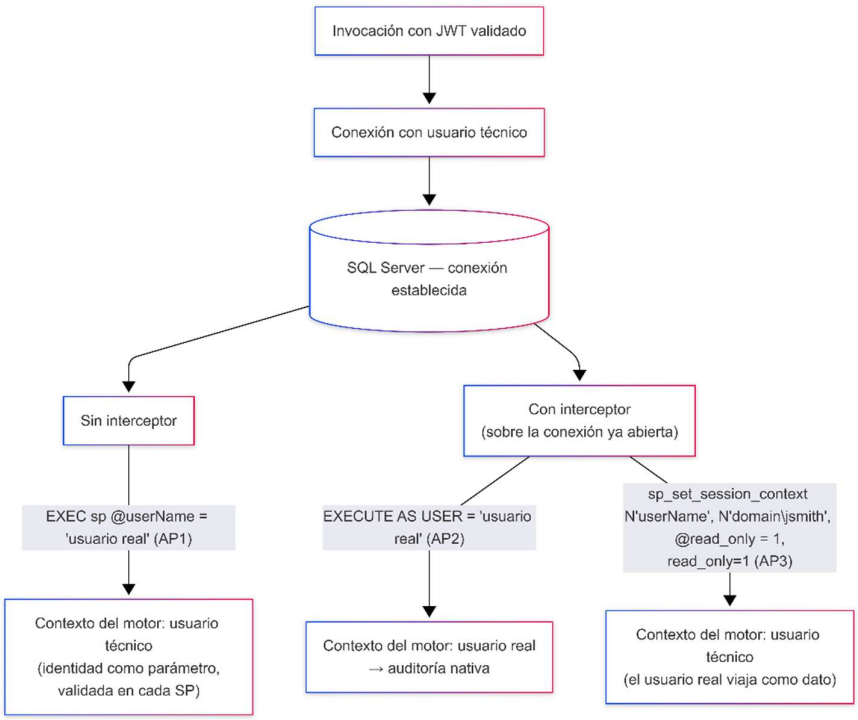

Fig. 1. Mecanismos de propagación de identidad hacia SQL Server. En los tres la conexión se establece con el usuario técnico. Sin interceptor, AP1 pasa la identidad real como parámetro; con interceptor, AP2 (EXECUTE AS) cambia el contexto al usuario real y AP3 (SESSION_CONTEXT) mantiene el usuario técnico y transporta la identidad como dato de sesión.

5 Análisis Comparativo

Tabla 1. Comparación de alternativas de conexión.

Atributo	AC1 (AD Svc)	AC2 (SQL Svc)	AC3 (SQL User)
Auditoría por usuario	No (nativa)	No (nativa)	Sí
Requiere dominio	Sí	No	No

Atributo	AC1 (AD Svc)	AC2 (SQL Svc)	AC3 (SQL User)
Complejidad operativa	Media	Media	Media
Containerizable	Limitado	Sí	Sí
Gestión de credenciales	AD Kerberos	Secret manager	Secret manager

Tabla 2. Comparación de alternativas de propagación de identidad.

Atributo	AP1 (Parámetro)	AP2 (EXECUTE AS)	AP3 (SESSION_CONTEXT)
Auditoría por usuario	Sí (aplicación)	Sí (motor nativo)	Sí (aplicación)
Requiere login SQL por usuario o grupo	No	Sí (individual o grupo de dominio)	No
Requiere refactor de SPs legacy	Sí (total)	No	Parcial
Compatible con usuarios no-AD	Sí	Solo con AC3	Sí
Riesgo de suplantación	Medio	Bajo	Bajo
Apto para migración progresiva	No	No	Sí

La combinación recomendada para escenarios de migración progresiva es **AC2 (SQL Login con secret manager) + AP3 (SESSION_CONTEXT)**, dado que:

1. No requiere que la infraestructura esté unida al dominio, habilitando containerización e infraestructura cloud.
2. No requiere un login SQL por usuario final, eliminando la complejidad de sincronización con el IdP.
3. No requiere refactorización completa de stored procedures legacy: solo los procedimientos que necesiten contexto de usuario deben actualizarse para leer el `SESSION_CONTEXT`.
4. Preserva el control de operaciones a nivel de stored procedure, manteniendo compatibilidad con el modelo de seguridad legacy durante la transición.

La combinación **AC1 o AC2 + AP2 (EXECUTE AS)** es preferible cuando la auditoría a nivel de motor es un requerimiento regulatorio explícito y el acceso directo a la base de datos ya ha sido eliminado por completo.

5.1 Aplicación en un caso real

El patrón AC2 + AP3 fue validado en un sistema de gestión de causas judiciales en producción (escenario híbrido de la Sección 3.3), donde aplicaciones de escritorio legacy con acceso directo a SQL Server conviven con una API REST basada en OIDC. Un interceptor sobre Entity Framework Core establece el SESSION_CONTEXT con el claim de usuario en cada apertura de conexión, y la identidad se resuelve en una única función centralizada (el único punto de modificación) que distingue usuarios de

dominio de usuarios técnicos. Ningún stored procedure requirió cambios, lo que confirma la viabilidad del patrón en un entorno con requisitos de trazabilidad judicial sin reemplazar la infraestructura legacy ni refactorizar el código existente.

6 Consideraciones de Seguridad para la Migración

6.1 Restricción de acceso de red

El principal objetivo de la migración es hacer de la API el único punto de acceso a los datos. Para lograrlo, el acceso TCP al puerto de SQL Server debe restringirse mediante reglas de firewall a únicamente las IPs de los servidores de aplicación. En entornos híbridos donde esto aún no es posible, la estrategia de `SESSION_CONTEXT` actúa como control compensatorio: los usuarios de dominio no tienen permisos DML directos sobre las tablas, solo `EXECUTE` sobre los stored procedures.

6.2 Resolución centralizada de identidad en escenarios híbridos

En un escenario híbrido, la identidad no debe derivarse directamente del `SESSION_CONTEXT` dentro de los stored procedures, sino resolverse en una única función centralizada: para la cuenta técnica de la API confía en el `SESSION_CONTEXT`; para una conexión directa de dominio lo ignora y usa el login real `SUSER_SNAME()`. Como corolario, los procedimientos que confían en el `SESSION_CONTEXT` deberían ser ejecutables solo por la cuenta técnica, de modo que ningún usuario de dominio disponga de un procedimiento donde un valor forzado tenga efecto. El `@read_only = 1` protege además la sesión de la API frente a inyección. El análisis del riesgo de suplantación que motiva estas medidas se desarrolla en la Sección 4 (AP3).

6.3 Gestión de credenciales y mínimo privilegio de las cuentas de base de datos

Las credenciales de la cuenta técnica (Alternativa de Conexión 2 (AC2)) no deben estar embebidas en el código fuente ni en archivos de configuración sin cifrar. Deben gestionarse mediante un secret manager (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault) con rotación automática y acceso auditado.

El principio de mínimo privilegio (Saltzer & Schroeder, 1975) debe aplicarse, y su alcance concreto difiere según el tipo de cuenta:

Cuentas de dominio (modelo legacy): deben tener únicamente permisos EXECUTE sobre los stored procedures necesarios, sin acceso DML directo a las tablas. La razón es estructural: un usuario de dominio puede conectarse a SQL Server con cualquier cliente externo (Management Studio, scripts, herramientas de terceros), por lo que restringir los permisos al nivel de procedimiento es la única barrera efectiva contra el uso indebido fuera del contexto de la aplicación.

Cuenta técnica SQL (AC2, modelo moderno): puede tener permisos DML sobre las tablas que accede, dado que esta cuenta no está expuesta a usuarios finales, solo es accesible a través de la API. Si la arquitectura migra hacia ORM o LINQ, el control de operaciones se desplaza completamente hacia el código de la API, y la base de datos deja de ser una capa de restricción. Esto implica que la superficie de riesgo ante una vulnerabilidad en la API es mayor, pero es el modelo esperado en una arquitectura moderna donde la API es el único punto de entrada.

Esta distinción es especialmente relevante en escenarios híbridos donde coexisten usuarios de dominio con acceso directo y la cuenta técnica de la API.

6.4 Revocación y expiración de sesiones

Una ventaja significativa del modelo OIDC sobre el modelo legacy es la capacidad de gestión centralizada del ciclo de vida de las sesiones. Los Access Tokens tienen expiración corta (típicamente 15-60 minutos). Al no existir credenciales de usuario a nivel de base de datos, la revocación del acceso (p. ej., baja de un usuario) puede efectuarse en el IdP sin necesidad de modificar logins en SQL Server.

6.5 Interacción con connection pooling

El SESSION_CONTEXT es un atributo de la sesión SQL Server, no de la conexión lógica de la aplicación. Como los drivers de acceso a datos (ADO.NET, EF Core) usan connection pooling por defecto, al "abrir" una conexión el driver puede entregar una sesión ya existente con su contexto previo intacto, propagando silenciosamente la identidad de un request anterior al siguiente.
Por eso el interceptor que llama a `sp_set_session_context` debe ejecutarse en cada apertura lógica de la conexión desde el pool (en EF, mediante `IDbConnectionInterceptor.ConnectionOpened`), no solo en la apertura física, y debe establecer siempre `@read_only = 1`: ante un intento de sobreescritura lanza una excepción (comportamiento fail-safe), garantizando que el `SESSION_CONTEXT` corresponda al usuario autenticado en el request actual.

7 Conclusiones

La migración hacia OIDC no resuelve por sí sola el problema de la identidad en la base de datos: lo desplaza. En organizaciones con sistemas legacy activos, el período híbrido concentra las debilidades de ambos modelos sin las ventajas de ninguno. Los patrones evaluados en este trabajo muestran que la brecha puede gestionarse sin requerir un reemplazo masivo de la infraestructura existente.
De las combinaciones evaluadas, AC2 + AP3 resulta la más viable para organizaciones en transición activa: no exige infraestructura de dominio, no requiere ninguna configuración de login en SQL Server por usuario o grupo, y permite migrar stored procedures de forma incremental. La auditoría por usuario, aunque no nativa en el

motor, queda garantizada a nivel de aplicación mediante el interceptor de conexión. Una vez completada la migración y eliminado el acceso directo a la base de datos, AC2 + AP2 (EXECUTE AS) representa el paso natural hacia auditoría nativa a nivel de motor, para organizaciones cuyos usuarios son de dominio o cuentan con login en SQL Server.

Una limitación abierta es el comportamiento del patrón en contextos de replicación y bases de datos distribuidas, donde el `SESSION_CONTEXT` no se propaga entre nodos y los agentes de replicación operan sin contexto de usuario.

8 Referencias

1. Bertino, E., & Sandhu, R. (2005). Database security, concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, *2*(1), 2–19. https://doi.org/10.1109/TDSC.2005.9
2. Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (pp. 13–15). IEEE.
3. Hardt, D. (Ed.). (2012). *The OAuth 2.0 Authorization Framework*. RFC 6749. Internet Engineering Task Force. https://doi.org/10.17487/RFC6749
4. Jones, M., Bradley, J., & Sakimura, N. (2015). *JSON Web Token (JWT)*. RFC 7519. Internet Engineering Task Force. https://doi.org/10.17487/RFC7519
5. Microsoft. (2024a). *SESSION_CONTEXT (Transact-SQL)*. Microsoft Learn. https://learn.microsoft.com/en-us/sql/t-sql/functions/session-context-transact-sql
6. Microsoft. (2024b). *EXECUTE AS Clause (Transact-SQL)*. Microsoft Learn. https://learn.microsoft.com/en-us/sql/t-sql/statements/execute-as-clause-transact-sql
7. Microsoft. (2024c). *sp_set_session_context (Transact-SQL)*. Microsoft Learn. https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-set-session-context-transact-sql
8. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., & Mortimore, C. (2014). *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Foundation. https://openid.net/specs/openid-connect-core-1_0.html
9. Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, *63*(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939