immoGenio

No. 01
Januar 2026

MULTI-TENANCY

der Hausverwaltungs-Software: Warum die Mandantentrennung in die Datenbank gehört, nicht in den Anwendungs-Code

Warum Mandantentrennung auf Datenbank-Ebene mit FORCE RLS gehört, nicht in den Anwendungs-Code: PostgreSQL Row-Level Security, Session-Kontext und das DSGVO-Versprechen.

AUTOR
ImmoGenio

VERÖFFENTLICHT
5. Januar 2026

ONLINE
www.immogenio.de/blog

— AUF EINEN BLICK

Inhalt

Das stille Versprechen jeder SaaS-Hausverwaltungs-Software

Wenn ein Anbieter mit dem Wort „mandantenfähig" wirbt, klingt das nach einer technischen Selbstverständlichkeit. In Wahrheit verbirgt sich dahinter eine der heikelsten Architektur-Entscheidungen, die ein SaaS-Hersteller treffen kann. Denn in den meisten gewachsenen Systemen teilen sich am Ende alle angeschlossenen Verwaltungen eine einzige physische Datenbank, oft sogar dieselbe Tabelle. Was die Mandanten voneinander trennt, ist eine einzige Spalte — `tenant_id` — und eine `WHERE tenant_id = ?`-Klausel, die der Anwendungscode bei jeder Datenbank-Abfrage selbst hinzufügen muss.

Solange jeder Endpunkt diese Klausel sauber setzt, funktioniert die Trennung. Vergisst aber ein Entwickler die Klausel in einem einzigen neuen Endpunkt — beim Schnellschuss kurz vor einem Release, im Reporting-Modul, in einer Migration, in einer Aggregations-Query —, sieht ein authentifizierter Nutzer der Verwaltung A plötzlich Datensätze der Verwaltung B. Das ist kein theoretischer Fall. Genau dieses Muster ist seit Jahren in CVE-Datenbanken dokumentiert und gehört zu den häufigsten Ursachen für Datenleaks in Multi-Tenant-SaaS.

Für eine Hausverwaltungs-Software ist das ein besonderes Problem. Die betroffenen Daten sind keine Marketing-Profile, sondern Mietverträge, Kontoauszüge, IBAN-Nummern, Kautionssalden, Eigentümeranschriften, Beschlussprotokolle. Eine Verwechslung zwischen zwei Verwaltungen ist nicht nur ein Reputationsschaden, sondern ein Verstoß gegen Art. 5 Abs. 1 lit. f DSGVO — die Pflicht zur Vertraulichkeit personenbezogener Daten — und gegen Art. 32 DSGVO, der dem Risiko angemessene technische und organisatorische Maßnahmen verlangt.

Dieser Beitrag beschreibt, warum die Mandantentrennung nicht im Anwendungs-Code, sondern in der Datenbank-Schicht stattfinden muss, was PostgreSQL Row-Level Security genau leistet, wo die typischen Fallstricke liegen und welche DSGVO-Aussage ein Verwalter erst dann ehrlich treffen kann, wenn die Trennung dort verankert ist, wo ein vergessener Filter sie nicht aushebeln kann.

Drei Modelle der Mandantentrennung — und ihre Ökonomie

Multi-Tenancy in SaaS-Systemen wird im Kern auf drei Wegen gelöst:

Eine **eigene Datenbank pro Mandant** ist die strengste Variante. Jede Verwaltung erhält eine physisch eigene PostgreSQL-Datenbank, oft sogar einen eigenen Server-Container. Backup, Migration und Skalierung erfolgen pro Tenant. Vorteil: Eine Verwechslung ist auf

Datenbankebene ausgeschlossen. Nachteil: Die Betriebskosten skalieren linear mit der Anzahl der Mandanten, jedes Schema-Upgrade muss N-mal durchgeführt werden, das Connection-Pooling wird komplex.

Ein **eigenes Schema pro Mandant** ist der Mittelweg. Die Datenbank ist gemeinsam, aber jeder Tenant hat ein eigenes PostgreSQL-Schema mit identischer Tabellenstruktur. Vorteil: Migrationen lassen sich teilautomatisieren. Nachteil: Cross-Tenant-Reporting wird aufwändig, das System muss Schema-Switching pro Request beherrschen.

Eine **gemeinsame Datenbank mit `tenant_id`-Spalte** ist der mit Abstand günstigste und gleichzeitig riskanteste Ansatz. Alle Mandanten liegen in denselben Tabellen, getrennt nur durch eine Discriminator-Spalte. Vorteil: Eine Migration, ein Backup, ein Connection-Pool. Nachteil: Ohne zusätzliche Schutzschicht hängt die gesamte Trennung an der Disziplin, mit der Anwendungs-Code geschrieben wird. Genau hier setzt Row-Level Security an — sie macht das dritte Modell sicherheitstechnisch wieder vertretbar.

Was Row-Level Security in PostgreSQL bedeutet

Row-Level Security, kurz RLS, ist ein PostgreSQL-Feature, das mit Version 9.5 im Jahr 2016 stabil wurde. Die Idee: Pro Tabelle definieren Sie eine oder mehrere `POLICY`-Regeln, die bei jeder Lese- oder Schreib-Operation angewendet werden. Eine Policy ist im Prinzip ein zusätzlicher SQL-Ausdruck, der für jede einzelne Zeile geprüft wird — gibt der Ausdruck `true` zurück, ist die Zeile sichtbar, sonst nicht.

Das entscheidende Detail steckt in zwei verwandten, aber unterschiedlichen Befehlen. `ALTER TABLE ... ENABLE ROW LEVEL SECURITY` aktiviert die Policies — aber nur für Nutzer, die nicht der Tabellen-Owner sind. Der Owner und alle Superuser umgehen die Policies. In einer typischen SaaS-Architektur ist genau das gefährlich: Wenn der Anwendungs-User aus historischen Gründen Owner der Tabelle ist, sieht er weiterhin alles.

`ALTER TABLE ... FORCE ROW LEVEL SECURITY` schließt diese Lücke. Mit FORCE werden die Policies auch für den Tabellen-Owner durchgesetzt. Nur Nutzer mit dem expliziten Attribut `BYPASSRLS` oder mit Superuser-Rechten können die Policies dann noch umgehen. Eine seriöse Multi-Tenant-Architektur verwendet ausschließlich FORCE RLS auf allen Tabellen mit `tenant_id`. Alles andere ist eine Ladung Glassplitter, die irgendwann zerbricht.

Eine typische Policy für eine Tabelle `mietvertraege` mit `tenant_id` als UUID-Spalte sieht so aus: Die Policy prüft, ob die `tenant_id` der Zeile gleich dem Wert ist, den die aktuelle Datenbank-Session in einem benutzerdefinierten Konfigurations-Parameter hinterlegt hat. Dieser Parameter wird in PostgreSQL über `current_setting('app.current_tenant')`

ausgelesen. Solange die Session diesen Parameter sauber setzt, sieht jede Query nur Zeilen des passenden Mandanten — egal, ob die Anwendung den `WHERE tenant_id = ?`-Filter mitgegeben hat oder nicht.

Wie der Session-Kontext sauber gesetzt wird

Der Punkt, an dem die meisten RLS-Implementierungen scheitern, ist der Übergang zwischen HTTP-Request und Datenbank-Transaktion. Der Tenant-Identifier kommt typischerweise aus einem JWT-Token oder einer Session — er muss aber bei der Datenbank ankommen, ohne dass der Anwendungs-Code in jeder einzelnen Query etwas tun muss.

Die saubere Lösung ist eine **Express-Middleware**, die für jeden Request eine Datenbank-Transaktion eröffnet, den Tenant-Identifier mit `SET LOCAL app.current_tenant = '...'` setzt und erst dann die eigentliche Route-Logik ausführt. `SET LOCAL` bedeutet: Der Wert gilt nur bis zum Ende der laufenden Transaktion und wird automatisch verworfen. Die Session „leakt" nicht in den nächsten Request, der den Connection-Pool-Slot übernimmt.

Innerhalb dieser Transaktion sind alle Queries auf den Tenant beschränkt — nicht durch Disziplin, sondern durch die Datenbank selbst. Auch eine vergessene `WHERE`-Klausel, ein neu hinzugefügter Endpunkt, eine Reporting-Query oder ein Aggregations-`COUNT(*)` über eine ganze Tabelle liefern nur die Zeilen des authentifizierten Mandanten. Der gesamte Klasse von „Entwickler-vergisst-Filter"-Bugs ist mit einem Schlag eliminiert.

BYPASSRLS und der App-User — der wichtigste Boot-Check

Eine RLS-Policy ist nur so stark wie der Datenbank-User, mit dem die Anwendung verbindet. PostgreSQL kennt drei kritische Eigenschaften, die alle Policies aushebeln: das `SUPERUSER`-Attribut, das `BYPASSRLS`-Attribut und der Status als Tabellen-Owner ohne FORCE RLS.

Eine ehrliche Architektur prüft beim Boot der Anwendung, mit welchem Datenbank-User sie verbunden ist und welche dieser Attribute er trägt. Production-Container müssen mit einem dezidierten App-User verbinden, der weder Superuser ist noch `BYPASSRLS` besitzt und nicht Owner der Multi-Tenant-Tabellen ist. Owner der Tabellen ist eine separate Rolle — etwa `app_owner` —, die nur für Migrationen verwendet wird und die in Production nicht für reguläre Requests freigeschaltet ist.

Die Konsequenz ist unbequem, aber notwendig: Wenn ein Hersteller in seiner Dokumentation schreibt „wir nutzen Row-Level Security" und gleichzeitig der App-User als Postgres-Superuser läuft, ist die Aussage technisch wertlos. Ein Audit, das diese Frage nicht stellt, ist ein oberflächliches Audit.

Tenant-Crossover-Tests gehören in die CI

Eine RLS-Architektur ist verifizierbar — und sollte es auch sein. Der Standard-Test, den jede Multi-Tenant-Software fahren muss, ist konzeptionell einfach. Zwei Tenants werden in der Test-Umgebung angelegt, jeder mit einigen Beispiel-Datensätzen. Ein Token für Tenant A wird ausgestellt. Mit diesem Token wird versucht, eine Ressource zu lesen, deren Identifier zu Tenant B gehört. Die erwartete Antwort ist nicht ein gefilterter Datensatz, sondern ein 404 — die Zeile existiert für diesen Request nicht, weil RLS sie ausblendet.

Derselbe Test wird für Schreib-Operationen wiederholt: Update auf Tenant-B-Ressource, Delete auf Tenant-B-Ressource, sowie für Aggregations-Endpunkte (`GET /api/mietvertraege` darf nur die Vertragszahl von Tenant A zurückliefern). Diese Tests laufen in der CI. Wenn ein Pull Request einen dieser Tests bricht, ist der Merge blockiert. Das ist keine Mehrarbeit — es ist die einzige Form, in der das Wort „mandantenfähig" überprüfbar wird.

Die DSGVO-Sicht: Art. 32, Risikoangemessenheit, BSI-Grundschutz

Art. 32 Abs. 1 DSGVO verlangt vom Verantwortlichen — also von der Hausverwaltung — „dem Risiko angemessene" technische und organisatorische Maßnahmen zum Schutz der Verarbeitung. Die Risikoangemessenheit ist keine abstrakte Kategorie, sondern bemisst sich an der Art der Daten, der Anzahl der Betroffenen und den potenziellen Schäden. Eine Hausverwaltung verarbeitet Mietverträge mit Geburtsdaten, IBANs, Bonitätsinformationen, Kautionsdaten und Beschlussprotokollen mit Eigentümernamen. Das ist ein Datenkatalog, der bei einer Verwechslung zwischen zwei Verwaltungen einen erheblichen materiellen und ideellen Schaden auslösen kann.

In dieser Risikoklasse ist eine reine Anwendungs-Code-Trennung nicht mehr als angemessen zu bezeichnen. Der BSI-Grundschutz-Baustein zur Mandantenfähigkeit fordert konsistent eine Trennung, die auch dann hält, wenn ein einzelner Layer der Anwendung kompromittiert oder fehlerhaft ist. Das deckt sich mit dem OWASP-Risiko A07:2021 (Identification and Authentication Failures), das genau die Klasse von Bugs adressiert, in der Authentifizierung formal stattfindet, der Zugriff auf Ressourcen aber nicht durchgehend autorisiert wird.

RLS verschiebt die Trennung in eine Schicht, die ein einzelner Anwendungsbug nicht aushebelt. Damit wird aus einer organisatorischen Disziplin eine technisch erzwungene Eigenschaft. Diese Differenz ist es, die ein DSGVO-Audit unterscheidbar macht: „Wir filtern in jeder Query" gegen „Die Datenbank weigert sich, Zeilen anderer Mandanten herauszugeben". Nur das zweite Versprechen ist überprüfbar.

Wie ImmoGenio die Trennung umsetzt

Die ImmoGenio-Plattform betreibt alle Multi-Tenant-Tabellen mit `FORCE ROW LEVEL SECURITY`. Eine zentrale Express-Middleware öffnet pro Request eine Transaktion und setzt `app.current_tenant` aus dem JWT-Token, bevor die Route-Logik beginnt. Die Datenbank-User der Production-Container haben weder Superuser-Status noch das `BYPASSRLS`-Attribut. Der Tabellen-Owner ist eine separate Migration-Rolle, die im laufenden Betrieb nicht verwendet wird.

Zwei Hardening-Migrationen sind in dieser Architektur besonders relevant. Migration 053 stellt sicher, dass Foreign-Key-Beziehungen zur `tenants`-Tabelle konsequent mit `ON DELETE CASCADE` versehen sind, sodass Tenant-Löschungen keine verwaisten Datensätze hinterlassen. Migration 056 schließt eine subtile Lücke, in der Snapshot-Trigger — also datenbankseitige Trigger, die historische Versionen schreiben — ohne korrekt gesetzten `app.current_tenant`-Kontext liefen. Solche Trigger benötigen eine eigene Policy-Berücksichtigung, weil sie technisch unter dem Trigger-Owner laufen.

Diese Architektur sitzt unter denselben offenen Schnittstellen, die die Plattform für DATEV-Export, Schließsystem-Integration und Messdienst-Daten bereitstellt — beschrieben in unserem Beitrag zu offenen API-Schnittstellen im Verwalter-Ecosystem. Jede Partner-API-Anfrage durchläuft denselben Tenant-Kontext-Mechanismus wie eine reguläre Web-Request. Es gibt keinen privilegierten API-Pfad, der die Mandantentrennung umgeht.

Der Edge-Case, der jede zweite Implementierung killt: Read-only-Pools

Sobald eine Anwendung wächst, kommt früher oder später der Wunsch nach einem dedizierten Read-only-Connection-Pool — etwa für Reporting, Dashboards oder Long-Running-Queries, die den Schreib-Pool nicht blockieren sollen. Hier wird RLS zur Stolperfalle. Wenn der Read-Pool den Session-Kontext nicht setzt, weil die Middleware nur den Schreib-Pool kennt, sieht der Read-Pool keine Daten — oder, schlimmer, er sieht alles, falls die Tabellen nicht mit FORCE RLS geschützt sind und der Pool-User Owner ist.

Die saubere Regel lautet: Jeder Connection-Pool, der gegen Multi-Tenant-Tabellen liest, **muss** denselben `SET LOCAL app.current_tenant`-Mechanismus durchlaufen. Wer einen Read-Pool ohne diesen Mechanismus betreibt, darf ihn nur für Daten verwenden, die explizit nicht tenant-scoped sind — etwa für globale Konfigurationswerte oder Lookup-Tabellen. Ein Verstoß gegen diese Regel ist die mit Abstand häufigste Ursache, warum eine theoretisch korrekte RLS-Architektur in Production löchrig wird.

Diagnostik: Wenn psql plötzlich „leere" Tabellen zeigt

Eine Begleiterscheinung von FORCE RLS ist, dass auch der entwickler-eigene psql-Zugriff in Production-Diagnostik blockiert ist. Wer mit dem App-User verbunden eine Query gegen eine Multi-Tenant-Tabelle absetzt und keinen Session-Kontext gesetzt hat, sieht null Zeilen. Der falsche Reflex lautet: „Die Tabelle ist leer, hier liegt ein Datenverlust vor". Der korrekte Reflex lautet: „RLS ist aktiv, ich habe keinen Tenant-Kontext, also sehe ich nichts".

Für legitime Cross-Tenant-Diagnostik existiert eine separate Admin-Rolle mit `BYPASSRLS`, die ausschließlich für Operations-Aufgaben verwendet wird, mit dediziertem Audit-Logging und außerhalb der regulären API-Pfade. Jede Verwendung dieser Rolle ist nachvollziehbar dokumentiert. Diese saubere Trennung — App-User ohne Bypass, Operations-User mit Bypass und Logging — ist die einzige Konstellation, die gleichzeitig sicher und betrieblich handhabbar ist.

Audit-Trail: Warum die Audit-Tabelle eigene Policies braucht

Die Audit-Log-Tabelle, die jede schreibende Aktion auf wesentlichen Entitäten dokumentiert, hat eine eigene Policy-Logik. Lese-Zugriffe sind tenant-scoped wie alle anderen Tabellen — eine Verwaltung sieht nur ihre eigenen Audit-Einträge. Schreib-Zugriffe sind erlaubt, Update- und Delete-Zugriffe aber nicht. Eine Audit-Zeile, die einmal geschrieben ist, kann von keinem Anwendungs-User mehr verändert werden. Diese Eigenschaft ist Pflicht für die Beweiskraft des Logs gegenüber Behörden und Eigentümern. Sie korrespondiert direkt mit den Anforderungen aus unserem Beitrag zu Aufbewahrungspflichten und revisionssicherem Archiv.

Die Grenzen — RLS ist kein RBAC

So wichtig RLS ist, sie ist nicht das Ende der Sicherheitsschicht. RLS schützt gegen Cross-Tenant-Verwechslungen. Sie schützt nicht gegen kompromittierte Admin-Accounts innerhalb eines Tenants — wenn ein Angreifer einen legitimen Admin-Token erbeutet, sieht er korrekt alle Daten dieses Tenants. RLS schützt auch nicht gegen logische Fehler in der Permission-Logik — wenn die Anwendung dem Buchhalter Zugriff auf Daten gewährt, die er nicht sehen sollte, sind die Policies korrekt, aber das Berechtigungssystem ist falsch konfiguriert.

Die Antwort auf diese Lücken ist ein zweites Schichtsystem: rollenbasierte Zugriffskontrolle (RBAC) mit feingranularen Permissions, MFA für privilegierte Accounts und ein konsistentes Audit-Log, das ungewöhnliche Zugriffe erkennt. RLS und RBAC sind komplementär — die eine schützt zwischen Tenants, die andere innerhalb des Tenants. Erst beide

zusammen ergeben die Architektur, in der eine Hausverwaltung mit gutem Gewissen ihren Mietern ein Selbstservice-Portal öffnen oder einen KI-Telefonassistenten anbinden kann, ohne die Vertraulichkeitsanforderungen aus Art. 5 DSGVO zu verletzen.

Wo wir stehen

Die ImmoGenio-Plattform betreibt heute alle Multi-Tenant-Tabellen produktiv mit `FORCE ROW LEVEL SECURITY`. Der Session-Kontext wird per Express-Middleware in jeder Request-Transaktion gesetzt. Die Production-DB-User sind weder Superuser noch tragen sie das `BYPASSRLS`-Attribut. Cross-Tenant-Tests laufen in der CI. Snapshot-Trigger und Audit-Tabellen sind mit eigenen Policy-Sets versehen. Die Migrationen 053 und 056 schließen die Lücken, die in der ersten Implementierung sichtbar geworden waren.

Das ist nicht der Endzustand — Sicherheit ist nie ein Endzustand —, aber es ist eine Architektur, die das Wort „mandantenfähig" hält. Eine Verwaltung, die ihre Daten in dieser Plattform betreibt, kann gegenüber dem eigenen Datenschutzbeauftragten nachweisen, dass die Mandantentrennung nicht eine Konvention im Anwendungs-Code ist, sondern eine technisch erzwungene Eigenschaft der Datenbank.

Kontakt

Wenn Sie als Verwalter oder als technisch verantwortlicher Entscheider in einem Verwaltungsbetrieb wissen wollen, wie diese Architektur in Ihrem konkreten Setup aussehen würde — etwa bei einer Migration aus einer bestehenden SaaS-Lösung oder bei einer DSGVO-Audit-Vorbereitung — sprechen Sie uns an. Wir zeigen Ihnen, welche Fragen Sie an Ihren aktuellen Anbieter stellen sollten und welche Antworten Sie verlangen dürfen.

ImmoGenio · kontakt@immogenio.de