immoGenio

No. 02
Mai 2026

RBAC

RBAC in der Hausverwaltungs-Software: Resource × Action statt Rollen-Paket — und warum „read_own" der wichtigste Action-Typ ist

Wie ein modernes RBAC-Modell in der Hausverwaltung aussieht: Resource × Action-Matrix, Verwaltungsbeirat-Rolle, read_own-Policies — und der DSGVO-Bezug zur Datenminimierung.

AUTOR
ImmoGenio

VERÖFFENTLICHT
5. Mai 2026

ONLINE
www.immogenio.de/blog

— AUF EINEN BLICK

Inhalt

Eine reale Verwaltung, eine alte Software, ein strukturelles Problem

Stellen Sie sich eine mittelgroße Hausverwaltung vor: sieben interne Mitarbeiter, drei externe Hausmeister auf Werkvertragsbasis, 1.200 Mieter über 92 Objekte verteilt, 280 Eigentümer in 14 WEG-Gemeinschaften, vier aktive Verwaltungsbeiräte. Jeder dieser Akteure hat einen anderen Informationsbedarf, eine andere rechtliche Stellung und eine andere Risiko-Klasse für personenbezogene Daten.

In der alten Software, die diese Verwaltung über Jahre eingesetzt hat, gibt es eine Rolle „Buchhalter" und eine Rolle „Mietverwalter". Wer Buchhalter ist, sieht alle Konten, alle Belege, alle Mahnungen, alle Telefonnummern aller Mieter aller Mandanten. Wer Mietverwalter ist, kann jeden Mietvertrag jedes Objekts öffnen — auch von Objekten, die er gar nicht betreut. Die Rolle ist ein Paket. Das Paket ist grob. Und genau dort beginnt das Problem, das ein modernes Rollen- und Berechtigungssystem lösen muss.

Dieser Artikel erklärt, warum das klassische Rollen-Paket-Modell in der Hausverwaltung an seine Grenzen stößt, warum eine Resource × Action-Matrix die saubere Antwort ist, was sich seit der WEG-Reform 2020 für den Verwaltungsbeirat technisch ändert und warum die Action `read_own` aus DSGVO-Sicht eigentlich der wichtigste Berechtigungs-Typ ist.

Warum grobe Rollen-Pakete im Verwaltungs-Alltag scheitern

Ein Rollen-Paket ist eine Sammlung fester Rechte unter einem Namen. „Buchhalter" enthält dann typischerweise: Belege erfassen, Zahlläufe ausführen, Mahnungen erstellen, Konten abstimmen, Darlehen verwalten, Auswertungen exportieren. Im Alltag entstehen aber regelmäßig Konstellationen, in denen diese Bündelung nicht passt:

- Ein Mandant möchte, dass ein neuer Buchhalter Belege buchen darf, aber keine Mahnungen verschickt — bis er das Mahnwesen verstanden hat.
- Eine externe Steuerberaterin soll Lese-Zugriff auf Konten und Journal bekommen, aber nichts schreiben.
- Ein Werkstudent soll Belege scannen und vorerfassen, jedoch keine Buchungssätze finalisieren.

Wenn die Software diese Differenzierung nicht abbildet, entsteht eine Workaround-Kultur: Mitarbeiter teilen Logins, Admins vergeben pauschal die volle Rolle und schicken eine E-Mail hinterher („bitte keine Mahnungen anfassen"), oder es entsteht ein Wildwuchs an

Sonder-Rollen wie „Buchhalter-Light", „Buchhalter-Lese", „Buchhalter-ohne-Mahnung". Nach 18 Monaten weiß niemand mehr, welche Rolle was wirklich darf. Genau das ist der Zustand, in dem Verwaltungen typischerweise zu uns kommen.

Das Resource × Action-Modell: feinkörnig, prüfbar, erweiterbar

Die Antwort darauf ist seit den 1990er-Jahren in der Sicherheits-Forschung etabliert. Sandhu, Coyne, Feinstein und Youman haben 1996 in „Role-Based Access Control Models" den Standard formuliert, der heute als NIST RBAC bekannt ist. Der Kern: Berechtigungen sind keine Rollen-Pakete, sondern Tupel aus Ressource und Aktion. Eine Rolle ist eine benannte Sammlung solcher Tupel — und kein in sich abgeschlossenes Paket.

Konkret in der Hausverwaltungs-Domäne sieht das so aus:

Resource beschreibt das fachliche Objekt, das geschützt wird:

`ANFRAGEN`, `BELEGE`, `DARLEHEN`, `EIGENTUEMER`, `EINHEITEN`, `HAUSMEISTER`, `JOURNAL`, `KAUTION`, `KONTENPLAN`, `KUENDIGUNGEN`, `MIETER`, `MIETVERTRAEGE`, `NEBENKOSTEN`, `OBJEKTE`, `OFFENE_POSTEN`, `RUECKLAGEN`, `USERS`, `VOICEBOT` und weitere.

Action beschreibt, was mit der Resource getan werden darf. Die Basis-Aktionen sind `read`, `create`, `update`, `delete`. Hinzu kommen domänenspezifische Aktionen wie `export`, `approve`, `validate`, `send`. Eine Permission ist dann ein Tupel — etwa `BELEGE × create`, `MAHNUNGEN × send` oder `JOURNAL × export`.

Eine Rolle wie `buchhalter` ist nun keine Black Box mehr, sondern eine konkrete Liste solcher Tupel. Sie können einer Rolle gezielt `BELEGE × create` geben, ohne `MAHNUNGEN × send` zu vergeben. Der Workaround-Druck verschwindet, weil die Software die feine Trennung von sich aus zulässt.

Die Spezial-Action `read_own`

Neben den klassischen Aktionen gibt es in einer Hausverwaltungs-Software eine Aktion, die auf den ersten Blick redundant wirkt, in Wahrheit aber den entscheidenden Unterschied macht: `read_own`. Sie sagt: „Lesen ist erlaubt — aber nur die eigenen Daten."

Für einen Mieter heißt das: nur die eigenen Mietverträge, die eigene Heizkostenabrechnung, die eigenen Schadensmeldungen. Für einen Eigentümer: nur die eigenen Einheiten, die eigenen Hausgeld-Abrechnungen, die eigenen Beschlüsse. Für einen Verwaltungsbeirat: nur die eigene WEG-Gemeinschaft, nicht die Nachbar-WEG, die zufällig im selben Mandanten liegt.

Aus dieser Aktion entsteht der gesamte Self-Service: das Mieter-Portal, das Eigentümer-Portal, der Beirats-Zugang. Wir haben den Self-Service in einem eigenen Artikel zum Mieter-Selbstservice im Detail beschrieben — der `read_own`-Mechanismus ist die technische Voraussetzung dafür, dass dieser Service überhaupt DSGVO-konform existieren kann.

Die Standard-Rollen einer Hausverwaltung

Die folgenden Rollen sind die typischen Ausgangsrollen, die in einem frisch aufgesetzten Mandanten existieren. Sie sind keine festen Pakete, sondern initiale Bündel, die jeder Mandant verfeinern kann.

- **admin** — der Tenant-Admin. Voller Zugriff auf alle Resources des Mandanten. Sieht Stammdaten, Buchhaltung, Verträge, kann Rollen vergeben, Benutzer einladen, Einstellungen ändern.
- **buchhalter** — Buchhaltungs-Funktionen. Belege, Journal, Konten, Offene Posten, Auswertungen. Kein Schreibzugriff auf Stammdaten wie Mieter, Eigentümer, Objekte.
- **mietverwaltung** — Mieter, Mietverträge, Übergaben, Kündigungen, Mietanpassungen, Indexierung. Lese-Zugriff auf Buchhaltung, kein Schreib-Zugriff.
- **weg_verwaltung** — WEG-spezifisch. Eigentümerversammlungen, Wirtschaftsplan, Vermögensbericht, Beschlusssammlung, Hausgeld.
- **hausmeister** — Tasks, Erledigungen, eigene Anfragen, eigene Stunden. Lese-Zugriff auf die Objekte der eigenen Zuständigkeit.
- **mieter** — ausschließlich `read_own`-Aktionen auf den eigenen Vertrag, die eigene Einheit, die eigenen Abrechnungen.
- **eigentuemer** — `read_own` auf die eigene Einheit, das zugehörige Objekt, die eigenen Hausgeld-Daten.
- **verwaltungsbeirat** — die Rolle, die seit der WEG-Reform 2020 fachlich neu zu denken ist. Erweiterte Lese-Rechte auf den eigenen WEG-Kontext, Vermögensbericht-Prüfung, Beschluss-Sammlung.

§ 29 WEG-Reform 2020 und der Verwaltungsbeirat

Mit der WEG-Reform vom 1. Dezember 2020 hat der Gesetzgeber die Rolle des Verwaltungsbeirats neu justiert. § 29 WEG legt fest, dass der Beirat den Verwalter unterstützt und überwacht. Konkret darf der Beirat:

- den Wirtschaftsplan vor Beschlussfassung prüfen und der Eigentümerversammlung eine Stellungnahme abgeben,
- die Jahresabrechnung prüfen,

- den Vermögensbericht einsehen und plausibilisieren.

Was der Beirat nicht darf: personenbezogene Daten anderer Eigentümer im Detail einsehen, soweit das nicht für die Prüfung erforderlich ist. Genau hier zeigt sich, warum eine grobe „Beirats-Rolle" das Problem nicht löst. Wenn die Software dem Beirat einfach Zugriff auf alle Eigentümer-Datensätze gibt, weil das technisch einfacher ist, verstößt das gegen die Datenminimierung aus Art. 5 Abs. 1 lit. c DSGVO.

Eine fachlich saubere Lösung bildet diese Differenz nicht durch eine Allzweck-Rolle ab, sondern durch konkrete `read`-Permissions auf den eigenen WEG-Kontext: `WIRTSCHAFTSPLAN × read` mit Read-Own-Scope auf die eigene Gemeinschaft, `VERMOEGENSBERICHT × read` mit demselben Scope, `BESCHLUSSSAMMLUNG × read` für die eigene WEG. Der Beirat sieht den Vermögensbericht, aber nicht die Kontoauszüge im Detail. Er sieht die Beschluss-Sammlung, aber nicht die Telefonnummer eines anderen Eigentümers. Wer das fachlich vertiefen möchte, findet im Artikel zur digitalen Eigentümerversammlung den passenden Workflow-Kontext.

Read-Own in der Praxis: Mieter sieht eigene Heizkostenabrechnung

Ein Mieter loggt sich in das Self-Service-Portal ein und ruft seine Heizkostenabrechnung ab. Technisch passiert dabei zweierlei: Erstens prüft die Anwendungsschicht, ob der angemeldete Benutzer die Permission `HEIZKOSTEN × read_own` besitzt. Zweitens setzt die Datenbank-Schicht eine Row-Level-Security-Policy um, die genau diese Zeile freigibt — und nur diese.

Die Policy auf der Tabelle `heizkosten_einzelabrechnungen` lautet sinngemäß: gib eine Zeile frei, wenn `mieter_id = current_setting('app.current_user.mieter_id')`. Der Mieter sieht seine eigene Abrechnung; die Abrechnung des Nachbarn taucht nicht einmal in der Antwort des Datenbank-Servers auf. Wer in der API einen Bug hätte und versehentlich „alle Heizkostenabrechnungen" abfragen würde, bekäme trotzdem nur die eigene Zeile zurück, weil die DB-Schicht den Filter erzwingt.

Diese zweite Schutzlinie — Row-Level-Security — beschreiben wir im Detail im Artikel zu Row-Level-Security und Multi-Tenancy. RBAC und RLS sind Geschwister, keine Konkurrenten: RBAC entscheidet, ob der Benutzer eine bestimmte Aktion grundsätzlich ausführen darf. RLS entscheidet, welche konkreten Zeilen er dabei zu Gesicht bekommt.

Permission-Auflösung in vier Schritten

Wenn ein API-Aufruf in das System hereinkommt, prüft die Anwendung die Berechtigung in einer festen Reihenfolge:

1. **Authentifizierung**: Ist der Benutzer überhaupt eingeloggt? Ist das JWT gültig? Gibt es eine aktive Session?
2. **Aktiver Tenant**: Welcher Mandant ist in der Session ausgewählt? Ein Buchhalter kann für mehrere Mandanten arbeiten — in jedem Request muss klar sein, in welchem Mandanten er gerade unterwegs ist.
3. **Effektive Rollen-Liste**: Welche Rollen hat der Benutzer in genau diesem Mandanten? Aus `user_roles` filtern wir nach `tenant_id` und bekommen die Liste der zugewiesenen Rollen.
4. **Resource × Action-Match**: Hat eine dieser Rollen die geforderte Permission? Aus `role_permissions` filtern wir nach den Rollen-IDs und prüfen, ob das gesuchte Tupel `(resource, action)` enthalten ist.

Damit dieser Vier-Schritt nicht in jedem Endpunkt zum Performance-Problem wird, laden wir die effektiven Permissions pro Request einmal über `loadUserPermissions()` in einen Request-Cache. Folge-Prüfungen innerhalb desselben Requests greifen auf den Cache zu, ohne weitere DB-Roundtrips. Auf der Frontend-Seite gibt es spiegelbildlich die Hooks `usePermission()` und `useAnyPermission()` sowie die Komponente `PermissionGate`, mit der Sie ganze UI-Bereiche, Buttons oder Menü-Einträge an Berechtigungen koppeln. Was der Benutzer nicht darf, sieht er auch nicht.

Drei Edge-Cases aus dem Hosted-Betrieb

Multi-Tenant-User. Ein externer Buchhalter arbeitet für drei WEG-Verwaltungen. In Mandant A hat er die Rolle `buchhalter`, in Mandant B die Rolle `buchhalter_lese`, in Mandant C die Rolle `admin`. Diese Zuordnungen sind in `user_roles` jeweils mit `tenant_id` gespeichert. Beim Tenant-Wechsel über den Tenant-Switcher wird die effektive Rollen-Liste neu berechnet. Es gibt keine globale „Super-Buchhalter"-Rolle, die Mandanten-übergreifend gilt — und das ist Absicht.

Sole-Admin-Schutz. Wenn der einzige Admin eines Mandanten gelöscht oder degradiert wird, wird die WEG verwaltungsunfähig. Niemand kann mehr Rollen vergeben, niemand kann mehr neue Benutzer einladen. Wir verhindern das durch einen Sole-Admin-Guard, der im Offboarding-Flow greift — beschrieben im Artikel zum User-Offboarding und Sole-Admin-Guard.

System-Account. Manche Operationen laufen ohne aktiven Benutzer — Cron-Jobs für Mahnläufe, ein Worker, der über Nacht Indexierungen berechnet, ein Webhook, der Banking-Daten importiert. Für diese Fälle gibt es einen System-Account mit eigener Permission-Liste. Im Audit-Trail erscheint nicht „User X hat Mahnung erzeugt", sondern „System-Mahnlauf 2026-04-15 hat Mahnung erzeugt". Das ist nachvollziehbar und prüfbar.

DSGVO: Datenminimierung als Default

Art. 5 Abs. 1 lit. c DSGVO verlangt Datenminimierung — personenbezogene Daten dürfen nur in dem Umfang verarbeitet werden, der für den Zweck erforderlich ist. Übersetzt in eine RBAC-Architektur heißt das: `read_own` ist der Default. Wer keine konkrete Permission auf eine Ressource hat, sieht von dieser Ressource nichts. Es gibt keinen impliziten Lese-Zugriff, kein „der hat ja sowieso Login, dann darf er auch alles im eigenen Mandanten sehen".

Art. 25 DSGVO — Privacy by Design and Default — geht einen Schritt weiter: die Trennung darf nicht erst durch Konfiguration entstehen, sondern muss strukturell eingebaut sein. Genau das leistet ein Resource × Action-Modell. Die Trennung ist in der Datenstruktur verankert, nicht in einer Rolle namens „Buchhalter", die in jedem Mandanten anders interpretiert werden könnte.

Auch § 535 BGB greift hier indirekt: das mietrechtliche Auskunftsrecht des Mieters auf die ihn betreffenden Abrechnungen wird durch `read_own`-Permissions technisch erfüllt — der Mieter sieht seine eigene Heizkostenabrechnung, ohne dass die Verwaltung ihm händisch ein PDF schicken muss.

Praxis: Ein Hausmeister, acht Objekte

Ein Hausmeister ist für acht Objekte zuständig. In der Software ist das hinterlegt — als Zuordnung in der Tabelle `hausmeister_objekte` oder vergleichbar. Wenn der Hausmeister sich anmeldet, sieht er in seiner Anfragen-Liste ausschließlich Anfragen aus diesen acht Objekten. Eine Anfrage aus einem neunten Objekt taucht in seiner Liste nicht auf — auch nicht in der Suche, auch nicht über einen Direkt-Link.

Technisch entsteht diese Sicht aus zwei Schichten. Die RBAC-Schicht erlaubt ihm `ANFRAGEN × read`. Die RLS-Schicht filtert die Zeilen so, dass nur Anfragen aus „seinen" Objekten zurückkommen. Der Hausmeister kann sich nicht „mal eben die anderen Objekte" anschauen, weil die Permissions zentral durch den Mandanten-Admin verwaltet werden — nicht durch den Hausmeister selbst.

Wenn der Hausmeister kündigt oder seine Zuständigkeit ändert, wird seine Rollen-Zuordnung angepasst. Ab diesem Zeitpunkt ist der Zugriff weg — sofort, ohne dass jemand „dran denken" muss, einen Login zu deaktivieren.

Wie ImmoGenio das technisch umsetzt

Auf der Datenbank-Ebene gibt es vier Tabellen, die das Modell tragen: `roles` definiert die Rollen pro Mandant, `permissions` listet die möglichen Resource × Action-Tupel, `role_permissions` verknüpft Rollen mit ihren Permissions, `user_roles` verknüpft Benutzer mit Rollen pro Mandant.

Auf der Anwendungsebene gibt es TypeScript-Enums für `Resource` und `Action`. Beide sind statisch typisiert — wer im Code eine nicht existierende Resource adressiert, bekommt einen Compile-Fehler. Im Frontend gibt es die Hooks `usePermission(resource, action)` und `useAnyPermission([...permissions])` sowie die Komponente `PermissionGate`, die UI-Bereiche an Permissions bindet. Auf der API-Seite prüft jeder geschützte Endpunkt im Middleware-Stack, ob die geforderte Permission vorhanden ist — nicht jeder Endpunkt einzeln, sondern deklarativ.

Migration `040_verwaltungsbeirat_role.sql` hat im Frühjahr 2025 die Verwaltungsbeirat-Rolle als initiale Rollen-Vorlage in das Baseline-Schema aufgenommen. Bei Mandanten-Anlage wird diese Rolle nicht automatisch zugewiesen — sie steht zur Verfügung, sobald ein WEG-Beirat im System angelegt und mit einem Login verknüpft wird.

Die natürlichen Nachbarn: RLS und Audit-Trail

RBAC ist nur ein Drittel des Sicherheits-Modells. Das zweite Drittel ist Row-Level-Security: die Antwort auf die Frage, welche Zeilen einem Benutzer zustehen, wenn er grundsätzlich lesen darf. Das dritte Drittel ist der Audit-Trail: jede Permission-Änderung, jede Rollen-Zuweisung, jede Rollen-Entziehung wird append-only protokolliert. Wer hat wann wem welche Rolle gegeben? Wer hat wann welche Permission entfernt? Diese Fragen müssen im Streitfall — und besonders im DSGVO-Audit — beantwortbar sein. Wir beschreiben das im Artikel zum Audit-Trail mit Append-Only-Mutationen.

Ergänzend kommt die Authentifizierungs-Schicht ins Spiel: Rollen-Zuweisungen mit weitreichenden Rechten — typischerweise `admin` und `buchhalter` — sollten an eine zweite Faktor-Schicht gekoppelt werden. Wir zeigen die Umsetzung im Artikel zur Multi-Faktor-Authentifizierung mit TOTP.

Was das Modell heute nicht kann

Wir bilden in der aktuellen Version reines RBAC ab — Benutzer haben Rollen, Rollen haben Permissions. Wir bilden noch kein vollständiges ABAC ab, also keine Attribut-basierten Zugriffe der Form „darf nur bearbeiten, wenn der Vertragsstatus aktiv ist und die letzte Mahnung mehr als 14 Tage zurückliegt". Solche Logik liegt aktuell in der Anwendungsschicht, nicht im Permission-System.

Wir bilden zudem keine zeitlich begrenzten Permissions ab — etwa „Beirat darf nur in der Vorbereitungswoche der Eigentümerversammlung den Wirtschaftsplan-Entwurf einsehen". Das ist ein Feature-Wunsch, der auf der Roadmap steht. Aktuell wird so etwas durch Rollen-Zuweisung und Rollen-Entziehung im Workflow abgebildet — also durch zwei diskrete Operationen, nicht durch ein Time-Window-Attribut.

Wo wir stehen

Das Resource × Action-Modell ist in ImmoGenio produktiv. Die Verwaltungsbeirat-Rolle ist als initiale Vorlage verfügbar. `read_own`-Policies sind für Mieter, Eigentümer und Beirat umgesetzt. Sole-Admin-Guard, System-Account und Multi-Tenant-Rollen-Trennung sind aktiv. Die `usePermission`-Hooks sind im gesamten Portal verfügbar, `PermissionGate` ist die einheitliche Komponente für UI-Gating.

Wenn Sie überlegen, wie die Berechtigungs-Architektur in Ihrer Verwaltung sauber aufgesetzt werden kann — gerade in Konstellationen mit Mehrfach-Mandanten, externen Buchhaltern oder einem aktiven Verwaltungsbeirat — schreiben Sie uns. Wir zeigen Ihnen die Standard-Rollen, das Resource × Action-Modell und die konkreten Read-Own-Policies in einer Sandbox-Umgebung Ihres eigenen Datenbestands.

Erreichbar unter kontakt@immogenio.de.