Engineer

RBAC

permissions granted to a role, not a person

What it is

RBAC decides what someone may do by looking at the roles they hold rather than at who they are. The letters stand for role-based access control, and NIST, where the model was formalised, also calls it “role based security”. A role is a named collection of permissions, such as “accounts clerk” or “read-only on production”. Users are assigned roles, and every permission a user exercises arrives through one of them.

David Ferraiolo and Rick Kuhn of NIST proposed the general model in 1992, as an alternative to the mandatory and discretionary access control of the day. In 1996 Sandhu, Coyne, Feinstein and Youman split it into layers: RBAC0 with users, roles and permissions, RBAC1 adding role hierarchies, RBAC2 adding constraints, and RBAC3 with both. The two lines were unified and became the standard ANSI/INCITS 359-2004, revised in 2012. Its reference model has four components. Core RBAC, the only one every system must have, holds the assignments of users to roles and permissions to roles, plus the session, in which a user activates a subset of the roles assigned. Hierarchical RBAC lets a senior role inherit a junior one. Static separation of duty constrains which roles one user may be assigned together, and dynamic separation of duty constrains which roles may be active at once in a session, so someone can hold both the role that raises a payment and the role that approves it without using both in one sitting.

The difference from an ordinary group is narrower than it looks. The NIST FAQ puts it this way: a group is usually a collection of users, and most group mechanisms also let a permission be attached straight to a user, which the FAQ calls a “loophole” that makes the user-permission relation hard to control. RBAC closes it by attaching permissions only to roles. The reason is in another line of the same FAQ: roles in an organisation stay fairly stable, whereas users and permissions are numerous and change quickly, so routing all access through roles makes it simpler to manage and to review.

Databases often implement the model directly. PostgreSQL has had one kind of entity since version 8.1, the role, which can act as a user, a group or both. A role meant as a group usually has no LOGIN, and membership in it is granted like any other privilege. Two membership options control how a member uses the group's privileges: INHERIT gives the member the group's privileges automatically, and SET lets the member switch into the role with SET ROLE, which comes close to the RBAC idea of activating a role only in the session that needs it.

```
-- the role carries the permissions; nobody logs in as it
CREATE ROLE reporting NOLOGIN;
GRANT USAGE ON SCHEMA sales TO reporting;
GRANT SELECT ON ALL TABLES IN SCHEMA sales TO reporting;

-- a person receives the role, never the table
CREATE ROLE alex LOGIN;
GRANT reporting TO alex;

-- a powerful role that has to be switched on deliberately
CREATE ROLE schema_admin NOLOGIN;
GRANT schema_admin TO alex WITH INHERIT FALSE;
-- later, in the one session that needs it: SET ROLE schema_admin;
```

Use the concept in a decision

Put "What is RBAC?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

Access control has to answer two questions that pull in different directions: is this request allowed right now, and who can do what across the whole system. Most schemes answer the first well. RBAC is built for the second, because its answer is a short list of roles and the people in each, which an auditor or a manager can actually read. NIST's one-line verdict on the alternative, attribute-based access control, where each decision evaluates rules over attributes of the user, the resource and the context, is that it "can be easy to set up but complex to manage".

The practical decision is where roles stop being enough. The 1992 model grants access “only if” the role requirements are met, a wording chosen to leave room for further checks at access time. A doctor role lets someone read medical charts, but not every patient’s chart: the relationship between this doctor and that patient is a fact about the data, not a job title. That is where a second mechanism belongs, such as row-level security in the database or an attribute rule in the application, layered on top of the roles rather than replacing them.

Our recommendation is to model access as roles first, named after jobs people actually do, with permissions attached only to roles and never directly to a login. Keep the role list short enough that one person can read all of it. Add row-level or attribute rules only where a role demonstrably cannot draw the boundary, and let the database enforce whatever it can, because a check made by the database applies to every client, including the code path someone forgot.

What it looks like in practice

In a database engineering job from 2022 to 2024, access to every developer environment, application instance, PostgreSQL and MS SQL database and Linux server came under 1,000 RBAC rules. The model was the simple part; the volume was the challenge. Permissions were broken into reusable categories such as “read-only access to production databases” and mapped to roles such as developers, application instances and database administrators. Developers reached only their designated servers, and application instances had limited permissions to prevent lateral movement. Ansible applied the rules consistently across environments, and the rules were documented so teams could audit or change them. The delivered result was the role-based rule set and a repeatable way to apply and review it.

Roles rarely work alone. When a map application moved from MSSQL to PostgreSQL with PostGIS, role-based access controls went in together with row-level security to enforce data-protection policies across several user groups. That pairing is the usual division of labour: roles decide which tables a group may touch, and row policies decide which rows inside them.

The strictest version takes the tables out of reach entirely. In a function-first PostgreSQL data layer from 2025, every operation the application performs is one of 1,275 functions across 34 schemas. The role the API logs in as, mariner, holds EXECUTE on the application functions and USAGE on the schemas and nothing else, which comes to roughly 4,000 explicit grants instead of one blanket

grant. The functions run as SECURITY DEFINER, owned by a separate non-login role with a pinned search_path, and the superuser account is kept for migrations and cron. A compromised API would still be limited to what the functions allow. The price is written down as well: a new field means a migration and a function change.

Where it goes wrong

- **Roles multiply until nobody can read them.** Every exception becomes a new role, such as “developer except billing” or “developer plus one report”, and soon there are more roles than people. Role explosion defeats the reason for RBAC, which was a list short enough to review. Give a person two or three roles rather than minting one per exception, and merge roles whose permission sets have become identical.
- **A permission lands on a login directly.** One urgent ticket, one GRANT straight to a user, and that user now has access that no role review will show. It is the loophole NIST describes in group systems, rebuilt by hand. Grant to roles only, and have the review query look for privileges held by roles that can log in.
- **Defaults hand out access before any role does.** In PostgreSQL the name PUBLIC means every role, and by default it receives EXECUTE on new functions and CONNECT and TEMPORARY on databases. Where a database was upgraded from PostgreSQL 14 or an older release, it can also create objects in the schema called public. Revoke those defaults deliberately, ideally inside the transaction that creates the object.
- **Roles only ever grow.** People change jobs and keep the old role beside the new one, and temporary access becomes permanent because removing it is nobody’s task. Each assignment needs a reason and a review date, and leaving a job should take away the roles that came with it.
- **Some accounts skip the rules.** Superusers and roles with BYPASSRLS ignore row-level security, and a table’s owner normally does too unless the table is set to FORCE ROW LEVEL SECURITY. An application that connects as the owner or as a superuser turns every row policy into decoration, so connect it as a role that owns nothing.

Further reading

- NIST Role Based Access Control project — the archived project home: the model’s history, the 2004 standard and its 2012 revision.
- NIST RBAC FAQ — roles versus groups, RBAC0 to RBAC3, separation of duty and the four components of the standard.

- PostgreSQL: Role Membership — group roles, granting and revoking membership, and the INHERIT and SET options.
- PostgreSQL: Row Security Policies — default deny, which roles bypass row policies, and FORCE ROW LEVEL SECURITY.
- PostgreSQL: Privileges — what PUBLIC receives by default and how to revoke it.
- SQL Server: Database-level roles — fixed and user-defined roles inside a SQL Server database.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is RBAC?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/rbac/