

# ADR-014: Per-Service Database Users

## Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

## Change Log

- approved 2026-02-15

## Referenced Use Case(s)

- CVP-ADR-014: Per-Service Database Users

## Context

The microservices architecture (ADR-008) deploys multiple services that all access a shared PostgreSQL database (ADR-002). The database access pattern chosen determines the blast radius if any single service is compromised. A service running with full database credentials could read voter data, alter election results, delete audit logs, or exfiltrate sensitive information regardless of what the application code is supposed to do.

The principle of least privilege states that each service should hold only the minimum permissions required for its function. In a shared-credential model every service has full access to every table, meaning a single compromised service exposes the entire database. Per-service PostgreSQL users restrict each service to the tables belonging to its domain, limiting the damage any one compromise can cause.

## Proposed Design

A dedicated PostgreSQL user is created for each service that requires database access. Each user is granted permissions only on the tables and operations that service needs, as defined by the domain boundaries established in ADR-013. The frontend service has no database user at all and communicates exclusively through backend service APIs.

## Permission matrix

Service	DB User	Tables Accessible	Permissions
auth-service	auth_user	organisers, voting_tokens, voter_mfa, blind_tokens, voters, elections, audit_log	SELECT, INSERT, UPDATE (specific tables)
election-service	election_user	elections, voters, election_options, voting_tokens, audit_log	SELECT, INSERT, UPDATE, DELETE
voting-service	voting_user	elections, election_options, encrypted_ballot s, vote_receipts, audit_log	SELECT, INSERT, UPDATE (blind_tokens only)

		blind_tokens, audit_log	
results-service	results_user	elections, election_options, encrypted_ballot s, tallied_votes	SELECT only on data tables; INSERT on tallied_votes
frontend-service	None	None	No database access (API consumer only)

All credentials are stored as Kubernetes Secrets and mounted as environment variables. They are never hardcoded in source files. The `deploy_platform.py` script creates database users with generated passwords and writes them to Secrets during deployment. The `schema.sql` file contains `CREATE USER` statements with placeholder passwords that are overridden at deployment time.

Each service maintains its own `asyncpg` connection pool with a minimum of 2 and maximum of 20 connections.

### Example grant statements

```

1 -- Auth service
2 CREATE USER auth_user WITH PASSWORD '<from-secret>';
3 GRANT SELECT, INSERT, UPDATE ON organisers TO auth_user;
4 GRANT SELECT ON voters, elections TO auth_user;
5 GRANT SELECT, UPDATE ON voting_tokens TO auth_user;
6 GRANT SELECT, INSERT ON voter_mfa, blind_tokens TO auth_user;
7 GRANT INSERT ON audit_log TO auth_user;
8 GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO auth_user;
9
10 -- Results service (read-only)
11 CREATE USER results_user WITH PASSWORD '<from-secret>';
12 GRANT SELECT ON elections, election_options, encrypted_ballots TO
    results_user;
13 GRANT SELECT, INSERT ON tallied_votes TO results_user;
14 GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO results_user;
15

```

Full grant statements for all users are maintained in the deployment script and in `ARCHITECTURE.md`.

### Considerations

#### Option 1: Shared database, shared credentials

All services use a single PostgreSQL user with full access to all tables. This was rejected because it provides no isolation at the database layer. A compromised results service could modify elections or delete audit logs. A compromised frontend service could query voter data directly. The approach undermines the security benefit of the microservice architecture entirely.

#### Option 2: Per-service PostgreSQL users (chosen)

Each service gets a dedicated user with grants scoped to its domain tables. The main drawback is credential management complexity: six users each require a password, and any schema change requires reviewing which users need updated grants. Both concerns are mitigated by the deployment script and the permission matrix documented here and in `ARCHITECTURE.md`.

#### Option 3: Fully separate databases

Each service runs its own PostgreSQL instance. This provides the strongest isolation but was rejected on resource grounds. Six PostgreSQL pods would consume roughly 3GB of RAM on a development machine with 16GB total. Distributed transactions would be required for any cross-service operation, and data shared between services (such

as election metadata) would need to be duplicated. The marginal isolation benefit over per-service users does not justify this cost for an MVP running on a local Kind cluster.

### Table-level vs row-level security

PostgreSQL Row-Level Security (RLS) would allow finer-grained control, for example restricting the election service to only the elections it created. This was deferred in favour of table-level GRANTS for the MVP. RLS will be evaluated at the end of Stage 2 if multi-tenant isolation becomes a requirement.

### Frontend isolation

The frontend service has no database user and no network path to PostgreSQL (enforced by the NetworkPolicy from ADR-010). This creates two independent layers of defence against SQL injection through the most externally exposed service.

### Defence in depth

Per-service database users, Kubernetes NetworkPolicies (ADR-010), and database-level immutability triggers (ADR-007) form three independent security layers. Each operates even if the others are bypassed.

### Decision

Per-service PostgreSQL users (Option 2) were chosen. The approach provides meaningful isolation at the database layer without the resource and operational overhead of separate database instances. If the voting service is compromised, the attacker gains INSERT access to encrypted ballots but cannot read voter email addresses, cannot modify election records, and cannot delete audit logs. The results service has read-only access regardless of application-level behaviour. The frontend service has no database access at all.

Schema changes that affect table ownership must include a review of the permission matrix and corresponding updates to grant statements. The permission matrix in this ADR and in [ARCHITECTURE.md](#) is the authoritative reference for what each service is allowed to do at the database level.

A review of whether Row-Level Security should be added is scheduled for the end of Stage 2 (April 2026).

### Other Related ADRs

- [ADR-002: PostgreSQL Database](#) - The shared database instance these users are created within
- [ADR-007: SHA-256 Hash-Chain Audit Logs](#) - Immutability triggers that complement permission restrictions on audit\_log and encrypted\_ballots
- [ADR-008: Microservices Architecture](#) - Each service uses its dedicated database user
- [ADR-010: Zero-Trust Network Policies](#) - NetworkPolicies restrict which services can reach PostgreSQL, providing a second layer of enforcement
- [ADR-011: Kubernetes Secrets](#) - Database credentials are stored and managed as Kubernetes Secrets
- [ADR-013: Domain-Driven Service Separation](#) - Service domain boundaries define which tables each user is granted access to

### References

- [PostgreSQL GRANT Documentation](#)
- OWASP Principle of Least Privilege
- Investigation Log, section 4.4: Full analysis of database access strategies