

ADR-002: PostgreSQL Database

Submitters

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

Change Log

- approved 2026-02-10

Referenced Use Case(s)

- UVote Database Requirements - Ballot integrity, encrypted ballot storage, audit log immutability, and per-service access control for the UVote election system.

Context

The UVote system requires a database that can guarantee ballot integrity through ACID transactions, provide native encryption primitives for ballot confidentiality, and enforce audit log immutability at the database level. The database must also support per-service access control to align with the microservices architecture, where each service should only access the tables relevant to its function.

The election domain has strict data integrity requirements. A lost or duplicated vote is a critical failure. Unlike an e-commerce system where a retry is acceptable, a double-counted ballot undermines the legitimacy of an entire election. This makes ACID compliance a hard requirement rather than a preference.

Early prototyping used SQLite for rapid iteration. While SQLite is effective for single-developer development, its file-level locking model does not support concurrent writes from multiple services. The project's security requirements (encrypted ballots, immutable audit logs, role-based access) also demand a database with server-grade security features that SQLite does not provide.

The selected approach is PostgreSQL 15, an open-source relational database with native support for all identified requirements via the pgcrypto extension, PL/pgSQL triggers, and a granular role-based access control system.

Proposed Design

Services and modules impacted:

All six UVote microservices connect to a single PostgreSQL 15 instance via the shared `database.py` `asyncpg` connection pool. Each service connects using a dedicated database user with least-privilege grants.

Connection pool configuration (`shared/database.py`):

```
1 import asyncpg
2
3 class Database:
4     _pool = None
5
6     @classmethod
7     async def get_pool(cls):
8         if cls._pool is None:
9             cls._pool = await asyncpg.create_pool(
10                 host=os.getenv("DB_HOST", "postgres-service"),
11                 port=int(os.getenv("DB_PORT", 5432)),
12                 database=os.getenv("DB_NAME", "uvote"),
13                 user=os.getenv("DB_USER"),
14                 password=os.getenv("DB_PASSWORD"),
15                 min_size=2,
16                 max_size=20,
17                 command_timeout=30
18             )
19
20         return cls._pool
```

Audit log immutability trigger (`schema.sql`):

```
1 CREATE OR REPLACE FUNCTION prevent_audit_modification()
2 RETURNS TRIGGER AS $$ 
3 BEGIN
4     RAISE EXCEPTION 'Audit log records cannot be modified or deleted';
5     RETURN NULL;
6 END;
7 $$ LANGUAGE plpgsql;
8
9 CREATE TRIGGER audit_immutability
10 BEFORE DELETE OR UPDATE ON audit_log
11 FOR EACH ROW
12 EXECUTE FUNCTION prevent_audit_modification();
13
```

Ballot encryption (`schema.sql`):

```
1 INSERT INTO ballots (election_id, voter_token_hash, encrypted_ballot)
2 VALUES (
3     $1,
4     $2,
5     pgp_sym_encrypt($3::text, current_setting('app.encryption_key'))
6 );
7
```

New modules:

No new shared modules are introduced. The existing `shared/database.py` is updated to configure the `asyncpg` connection pool with the parameters above.

Model and schema impact:

The schema is defined in `schema.sql` and includes table definitions, indexes, triggers, and per-service GRANT statements. JSONB columns are used for election configuration with GIN indexes to support efficient queries on election metadata.

Configuration:

- PostgreSQL version: 15 (official Docker image: `postgres:15`)
- Extensions: `pgcrypto` (`CREATE EXTENSION IF NOT EXISTS pgcrypto`)
- Connection pool: `min_size=2` , `max_size=20` , `command_timeout=30s`
- Storage: PersistentVolumeClaim (1Gi, ReadWriteOnce)

Integration points:

- Auth service: users table (INSERT, SELECT, UPDATE) via `auth_user` role
- Voting service: ballots table (INSERT only) via `voting_user` role
- Results service: ballots table (SELECT with `pgp_sym_decrypt`) via `results_user` role
- Election service: elections table (full CRUD) via `election_user` role
- Kubernetes: PostgreSQL Deployment, Service, and PVC manifests (see ADR-003)
- Secrets: database credentials and encryption key stored as Kubernetes Secrets (see ADR-003)

DevOps impact:

The PostgreSQL instance runs as a single pod in the Kubernetes cluster with a PersistentVolumeClaim for data durability. The schema is initialised from `schema.sql` on first startup. Schema changes are applied manually via `psql` during development; a formal migration tool such as Alembic may be introduced in Stage 2 if schema churn increases.

Considerations

MySQL 8.0 (rejected): MySQL is familiar from DkIT coursework and provides ACID compliance via the InnoDB engine. It was rejected primarily because it has no equivalent to pgcrypto for in-database encryption. MySQL's `AES_ENCRYPT` function exists but lacks PGP-grade primitives, meaning ballot encryption would have to be handled entirely at the application layer. MySQL's trigger system is also less expressive than PL/pgSQL for audit enforcement, and its per-user privilege model is less granular than PostgreSQL's role-based system. The `asyncmy` driver is less mature and performant than `asyncpg`.

MongoDB 7.0 (rejected): MongoDB provides flexible schema and native document storage. It was rejected on three grounds. First, multi-document ACID transactions were added in v4.0 but carry a documented performance penalty. Second, in-database encryption requires MongoDB Enterprise, which is a paid product. Third, change streams are reactive rather than preventive and cannot enforce audit immutability in the same way as a PL/pgSQL trigger. More fundamentally, the election data model is relational by nature: voters are registered for elections, ballots reference candidates, and results aggregate votes. Representing this in a document model adds complexity without benefit.

SQLite 3 (rejected): SQLite was used for early prototyping due to its zero-configuration setup. It is unsuitable for production use in this system because its file-level locking permits only one writer at a time, which is incompatible with concurrent ballot submissions from multiple services. SQLite also has no user or role system, no native encryption extension in its standard build, and no support for `asyncpg`. It is not suitable for multi-service access across containers.

Operational complexity: PostgreSQL requires server process management, user creation, and extension installation. This is accepted as a one-time setup cost given the security requirements it satisfies.

Single point of failure: One PostgreSQL instance serves all six microservices. If the database becomes unavailable, all services are affected. This is mitigated by Kubernetes health probes and PVC-backed storage, which ensures data survives pod restarts. A multi-node setup is out of scope for this project.

Key management: pgcrypto's symmetric encryption requires the encryption key to be distributed securely. The key is stored as a Kubernetes Secret and injected as an environment variable. A compromise of the Secret would compromise ballot confidentiality. This is an accepted risk within the project's scope.

Decision

PostgreSQL 15 was selected as the database for all UVote microservices.

PostgreSQL is the only option evaluated that satisfies all security-critical requirements: native ballot encryption via pgcrypto, audit log immutability via PL/pgSQL triggers, and per-service user roles with least-privilege grants. These controls are enforced at the database layer, which provides a stronger security posture than application-level enforcement alone. A compromised service cannot bypass database-level triggers or exceed its granted privileges.

The three security factors were the primary drivers. The `pgp_sym_encrypt` function encrypts ballot content within the database using AES-256, meaning ballot plaintext is not persisted in unencrypted form. The `BEFORE DELETE OR UPDATE` trigger on the `audit_log` table raises an exception for any attempt to modify or delete audit records, making the log append-only regardless of which service is making the request. Per-service database users are granted only the privileges required for their function: the voting service can insert ballots but cannot read them, and the results service can read and decrypt ballots but cannot modify them.

JSONB columns with GIN indexing allow election configuration to evolve without schema migrations, which suits iterative development.

The trade-off of increased operational complexity over SQLite is accepted. The setup cost is a one-time investment; the security guarantees it provides are required for the lifetime of the project.

A review is scheduled for the end of Stage 2 (April 2026) to assess whether the single-instance setup meets scaling requirements or whether connection pooling via PgBouncer is needed.

Other Related ADRs

- ADR-001: Python FastAPI Backend - asyncpg dependency
- ADR-003: Kubernetes Platform - PersistentVolumeClaim and Secrets management
- ADR-005: Token-Based Voting - ballot storage design

References

- [PostgreSQL 15 Documentation](#)
- [pgcrypto Documentation](#)
- [asyncpg Documentation](#)
- [EdgeX Foundry ADR Template](#)