Design and Implementation of ebuild-repo Project

Ali Efruz YILDIRIR (64230018)

January 27, 2026

Abstract

`ebuild-repo` is a lightweight backend service that provides a RESTful API and a grounded relational storage layer for publishing, searching, and versioning ebuild package metadata. The system models packages, maintainers, versions, and token-audit records in a normalized SQL schema optimized for read-heavy queries and straightforward migration (see the migrations directory). The database layer encapsulates persistence and transactional logic, exposing concise interfaces for CRUD operations, version history, and indexing of searchable fields.

The RESTful API follows resource-oriented principles: endpoints accept and return JSON representations of packages, versions, and user actions, use JWT-based authentication for protected routes, and implement pagination and filtering for efficient discovery. Endpoints are mapped to handler functions which validate input, orchestrate store transactions, and translate domain errors into HTTP status codes. Together, the database schema and API contract prioritize data integrity, auditability, and predictable performance for common operations (publish, update, search, and fetch version manifests), enabling integration with web clients and automated tooling that manage ebuild lifecycle at scale.

The RESTful API is modelled for usage in a `cli` application. But the project also supplies a very basic web frontend that supports the login, register, logout, package, version, artifact operations.

Many C/C++ projects rely on scattered build scripts, ad-hoc metadata files, or private registries, which complicates consistent versioning, reproducible builds, and automated discovery. ebuild-repo centralizes build manifests and version metadata for a custom C/C++ build system (namely the repo in github yldrefruz/ebuild), providing a single, auditable source of truth with explicit version history and token-audit trails. By exposing this data via a well-defined RESTful API, the service reduces synchronization errors between maintainers and automation, enables efficient search and filtering of build artifacts, and supplies the authentication and audit logging required for secure, automated build and deployment pipelines.

1 Framework

The project utilizes many different libraries and other thirdparty software for managing the database. Since developers should be able to iterate quickly on the project the project uses sqlite3 in a development environment meanwhile using postgresql in a release environment. The connection to the databases are managed through a connection string that can be specified as an environment variable or in the config.

1.1 pressly/goose

Goose is a database migration tool. It is written in go and supports sql migrations with annotations[3]. Migrations with these tools should be written for bot directions. An up section looks like this:

Listing 1: goose up migration

```sql
-- +goose Up
CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT NOT NULL UNIQUE,
email TEXT NOT NULL UNIQUE,
password_hash TEXT NOT NULL,
role TEXT NOT NULL DEFAULT 'public',
created_at DATETIME DEFAULT
CURRENT_TIMESTAMP
);
```

Meanwhile a down section looks like this:

Listing 2: goose down migration

```sql
-- +goose Down
DROP TABLE IF EXISTS packages_fts;
DROP TABLE IF EXISTS tokens;
DROP TABLE IF EXISTS comments;
DROP TABLE IF EXISTS votes;
DROP TABLE IF EXISTS package_buckets;
DROP TABLE IF EXISTS package_categories;
DROP TABLE IF EXISTS buckets;
DROP TABLE IF EXISTS categories;
DROP TABLE IF EXISTS packages;
DROP TABLE IF EXISTS users;
```

Database is migrated with a command like this:

Listing 3: output of the goose up command

```
$ goose up
  OK    001_basics.sql
  OK    002_next.sql
  OK    003_and_again.go
```

this applies only the sections with `-- +goose Up`. The state of the database is stored inside the database. It stores the applied migrations inside of a table. Then based on these migrations, the database is updated.

1.2 SQLite

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day[4].

This library has bindings for multiple programming languages. go is one of them, so I used the binding `https://github.com/mattn/go-sqlite3`. SQLite is used only in development to allow fast-iterations and quick inspections. Program still uses PostgreSQL in production.

1.3 PostgreSQL

PostgreSQL is world's most advanced open source relational database. It is powerful and efficient[2]. Thjs database is used only in production mode since it requires a somewhat complex installation and is hard to inspect for what is happening in the background. Also PostgreSQL has features that SQLite doesn't have so some of the features unfortunately are not used.

1.4 golang

Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language[1].

This programming language is mainly selected because its support for the framework elements listed above and it being compiled to native code instead of being interpreted.

2 Initial Tables

The database is initialized with the below code.

2.1 Users Table

The user table describes what an user is, stores the hash of their password, role, creation time, etc. The table can still be expanded to include various other data. For example a user can be deleted or banned permanently. But for this implementation of package manager we don't need this for this development stage. Columns can still be added with creating a new migration

Listing 4: users

```
CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT NOT NULL UNIQUE,
email TEXT NOT NULL UNIQUE,
password_hash TEXT NOT NULL,
role TEXT NOT NULL DEFAULT 'public',
created_at DATETIME DEFAULT
 CURRENT_TIMESTAMP
);
```

id the unique id of the user. This is the actual identifier of the user.

username here stored as text must be unique. But it would actually be better to store as VARCHAR(32) since the storage of `TEXT` is virtually infinite.

email the mail of the user, must be unique. If not unique the database will error and the user will see that this email is used. The storage of text type is wrong, it should be VARCHAR(128)

password_hash hash of the password user passed in with a salt. The password itself is stored as hash since in the case of a leak, the hash will leak not the actual password of the user, this even though not fully secure, gives a little bit of a security.

role the role of the user. May be admin or public. If the role is admin, the user can change any packages information or artifacts.

created_at the creation date of the user's account.

2.2 Packages Table

The packages table stores metadata about packages published in the registry: name, description, creator, token requirement and creation time.

Listing 5: packages

```
CREATE TABLE IF NOT EXISTS packages (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL UNIQUE,
description TEXT,
created_by INTEGER NOT NULL REFERENCES
 users(id) ON DELETE SET NULL,
token_required INTEGER NOT NULL DEFAULT 1,
created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

id the unique identifier for the package.

name package name, must be unique.

description optional textual description of the package.

created_by references users(id); identifies the creator/owner. Note: the schema sets NOT NULL together with ON DELETE SET NULL which is contradictory — make this column nullable or change the foreign-key action.

token_required boolean-like flag (stored as INTEGER) indicating whether a token is required to publish/manage the package (1 = required, 0 = not required).

created_at timestamp when the package record was created (defaults to CURRENT_TIMESTAMP).

2.3 Categories Table

Stores package categories used to organize packages.

Listing 6: categories

```
CREATE TABLE IF NOT EXISTS categories (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL UNIQUE,
description TEXT
);
```

id unique identifier for the category.

name category name, must be unique.

description optional description.

2.4 Buckets Table

Logical groupings (buckets) for packages.

Listing 7: buckets

```
CREATE TABLE IF NOT EXISTS buckets (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL UNIQUE,
description TEXT
);
```

id unique identifier for the bucket.

name bucket name, must be unique.

description optional description.

2.5 Package-Categories Join Table

Many-to-many relation between packages and categories.

Listing 8: package categories

```
CREATE TABLE IF NOT EXISTS
package_categories (
package_id INTEGER NOT NULL
 REFERENCES packages(id) ON
  DELETE CASCADE,
category_id INTEGER NOT NULL
 REFERENCES categories(id)
  ON DELETE CASCADE,
PRIMARY KEY (package_id, category_id)
);
```

package_id references packages.id; cascade on package deletion.

category_id references categories.id; cascade on category deletion.

PRIMARY KEY composite key prevents duplicate assignments.

2.6 Package-Buckets Join Table

Many-to-many relation between packages and buckets.

Listing 9: package buckets

```
CREATE TABLE IF NOT EXISTS
 package_buckets (
package_id INTEGER NOT NULL
 REFERENCES packages(id) ON
  DELETE CASCADE,
bucket_id INTEGER NOT NULL
 REFERENCES buckets(id) ON
  DELETE CASCADE,
PRIMARY KEY (package_id, bucket_id)
);
```

package_id references packages.id; cascade on package deletion.

bucket_id references buckets.id; cascade on bucket deletion.

PRIMARY KEY composite key prevents duplicate assignments.

2.7 Votes Table

Stores user votes for packages (one vote per user per package).

Listing 10: votes

```
CREATE TABLE IF NOT EXISTS
 votes (
id INTEGER PRIMARY KEY
 AUTOINCREMENT,
user_id INTEGER NOT NULL
 REFERENCES users(id) ON
  DELETE CASCADE,
package_id INTEGER NOT NULL
 REFERENCES packages(id) ON
  DELETE CASCADE,
value INTEGER NOT NULL,
created_at DATETIME DEFAULT
 CURRENT_TIMESTAMP,
UNIQUE(user_id, package_id)
);
```

id vote id.

user_id voter, cascades on user deletion.

package_id target package, cascades on package deletion.

value numeric vote value (e.g. -1/1 or score).

created_at timestamp.

UNIQUE prevents multiple votes by the same user for a package.

2.8 Comments Table

User comments on packages (optionally tied to a package version).

Listing 11: comments

```
CREATE TABLE IF NOT EXISTS comments (
id INTEGER PRIMARY KEY AUTOINCREMENT,
user_id INTEGER NOT NULL REFERENCES
 users(id) ON DELETE CASCADE,
package_id INTEGER NOT NULL REFERENCES
 packages(id) ON DELETE CASCADE,
package_version_id INTEGER NULL,
body TEXT NOT NULL,
created_at DATETIME DEFAULT
 CURRENT_TIMESTAMP
);
```

id comment id.

user_id author, cascades on user deletion.

package_id related package, cascades on package deletion.

package_version_id optional link to a package version.

body comment text.

created_at timestamp.

2.9 Tokens Table

API/auth tokens owned by users.

Listing 12: Tokens

```
CREATE TABLE IF NOT EXISTS tokens (
id INTEGER PRIMARY KEY AUTOINCREMENT,
owner_user_id INTEGER NOT NULL
 REFERENCES users(id) ON DELETE CASCADE,
token_hash TEXT NOT NULL,
is_generated INTEGER NOT NULL DEFAULT 0,
scopes TEXT,
allowed_package_ids TEXT,
revoked_at DATETIME NULL,
created_at DATETIME DEFAULT
 CURRENT_TIMESTAMP
);
```

id token id.

owner_user_id owner user, cascades on user deletion.

token_hash hashed token value.

is_generated boolean-like flag for generated tokens.

scopes text list of scopes.

allowed_package_ids text list (e.g. CSV/JSON) limiting package access.

revoked_at nullable revocation timestamp.

created_at creation timestamp.

2.10 Packages Full-Text Search (FTS)

Virtual table for searching package name/description via FTS5. This is an extension for SQLite, not available on all distributions of SQLite.

Listing 13: FTS5 virtual table

```
CREATE VIRTUAL TABLE IF NOT
 EXISTS packages_fts USING fts5(
name, description,
 content='packages', content_rowid='id'
);
```

FTS table provides fast full-text searching on packages.name and packages.description.

content/content_rowid linked to the packages table; syncing may require triggers or manual updates.

3 Extensive view of the API and Database Queries

The RESTful api and the database works together. So in this section the two will be explained together.

3.1 Server bootstrap and runtime configuration

The server entrypoint is implemented in the file `cmd/server/main.go`. In development mode the project uses SQLite via `github.com/mattn/go-sqlite3`. The database location is controlled by the environment variable `DEV_DB` (defaults to `dev.db`).

Listing 14: Development server bootstrap (SQLite + Gin)

```
dbPath := os.Getenv("DEV_DB")
if dbPath == "" { dbPath = "dev.db" }
db, err := sqlx.Open("sqlite3", dbPath)

signingKey := []byte("dev-signing-key")
r := api.SetupRouter(db, signingKey)
r.Run(":8080")
```

3.2 Routing overview (Gin)

HTTP endpoints are registered in `internal/api/router.go`. Each route is mapped to a handler function (usually one file per feature). Protected routes are enforced using JWT scopes via `RequireScope`.

Method	Path	Purpose
GET	/health	Health check (always returns `{"status":"ok"}`).
POST	/register	Create a user account.
POST	/login	Authenticate, return access token, set refresh + CSRF cookies.
POST	/refresh	Rotate refresh token (cookie-based) and issue new access token.
GET	/me	Return current user identity (JWT required).
POST	/packages	Create a package (requires `maintain` scope).
GET	/packages/:id	Fetch package + latest version (if any).
POST	/packages/:id/versions	Publish a new version (requires maintainer/admin).
GET	/packages/:id/versions	List versions (newest first).
POST	/packages/:id/versions/:ver/artifacts	Add artifact metadata (requires maintainer/admin).
GET	/artifacts/:artifact_id/download	Redirect to blob URL and increment download counters.
POST	/packages/:id/votes	Upsert a vote (JWT required).
POST/GET	/packages/:id/comments	Create/list comments.
GET	/search	Search packages (FTS5 if available, otherwise LIKE fallback).

Table 1: Primary API routes from `internal/api/router.go`.

3.3 Middleware: authentication, revocation, and CSRF

All requests pass through two middlewares in `internal/api/middleware.go`:

a) **Authentication:** If the request has an `Authorization: Bearer <jwt>` header, the server parses the token, then checks the `tokens` table to see if the token has been revoked.

b) **CSRF protection:** For state-changing requests (POST/PUT/PATCH/DELETE) the server enforces `X-CSRF-Token` matching the `ebuild_csrf` cookie, unless an Authorization header is present (API clients), or the path is exempted (login/register/refresh/revoke).

Revocation check query The revocation model stores only a SHA-256 hash of the raw token and checks `revoked_at`.

Listing 15: Token revocation check (AuthMiddleware)

```sql
SELECT revoked_at
FROM tokens
WHERE token_hash = ?
LIMIT 1;
```

3.4 Authentication and token lifecycle

The authentication flow is implemented in `internal/api/handlers_auth.go`, `internal/api/handlers_refresh.go`, and `internal/api/handlers_tokens.go`.

Register Passwords are stored as bcrypt hashes. New users default to the maintainer role.

Login Creates a short-lived access token (JWT) and a long-lived refresh token (JWT). The refresh token is stored as an *HTTP-only cookie* and also recorded (hashed) in the database for revocation/audit. A second non-HTTP-only cookie stores the CSRF token.

Refresh Rotates refresh tokens: the old token is revoked and a new token is issued. This creates an explicit parent → child relationship through `parent_token_hash`.

Generated tokens Users can create long-lived tokens with explicit scopes (stored in `tokens.scopes`).

Listing 16: User register and login queries

```sql
INSERT INTO users (username, email, password_hash,
    role)
VALUES (?, ?, ?, ?);

SELECT id, username, email, password_hash, role
FROM users
WHERE username = ?;
```

Listing 17: Refresh token persistence and rotation

```sql
INSERT INTO tokens (owner_user_id, token_hash,
    is_generated, scopes, created_at)
VALUES (?, ?, 0, ?, datetime('now'));
```

```
UPDATE tokens
SET revoked_at = datetime('now')
WHERE token_hash = ?;

INSERT INTO tokens (owner_user_id, token_hash,
     is_generated, scopes, parent_token_hash,
     created_at)
VALUES (?, ?, 0, ?, ?, datetime('now'));
```

Audit trail (token actions) Token creation and revocation events are written to `token_audit`. This provides an append-only history useful for incident response and debugging.

Listing 18: Token audit writes

```
INSERT INTO token_audit (action, token_hash,
     owner_user_id, actor_user_id, parent_token_hash
     , meta)
VALUES (?, ?, ?, ?, ?, ?);

INSERT INTO token_audit (action, token_hash,
     owner_user_id, actor_user_id)
VALUES (?, ?, ?, ?);
```

3.5 Packages and versions

Package operations are implemented in `internal/api/handlers_package.go` and `internal/api/handlers_versions.go`. Packages can be created by any authenticated user with the `maintain` scope; publishing versions additionally requires the caller to be the package owner, a listed maintainer, or an admin.

Listing 19: Package create + read queries

```
INSERT INTO packages (name, description, created_by,
      token_required)
VALUES (?, ?, ?, 1);

SELECT id, name, description, created_by,
     token_required, created_at
FROM packages
WHERE id = ?;

SELECT id, package_id, version, metadata,
     released_by, released_at, is_deprecated
FROM package_versions
WHERE package_id = ?
ORDER BY released_at DESC
LIMIT 1;
```

Maintainer/admin authorization checks Authorization is done with lightweight point-queries:

Listing 20: Maintainer/admin authorization queries

```
SELECT role
FROM users
WHERE id = ?;

SELECT created_by
FROM packages
WHERE id = ?;

SELECT 1
FROM package_maintainers
WHERE package_id = ? AND user_id = ?
LIMIT 1;
```

Publish and list versions Version strings are validated using semantic versioning (`github.com/Masterminds/semver`) before inserting.

Listing 21: Version publish and listing queries

```
INSERT INTO package_versions (package_id, version,
     metadata, released_by, released_at)
VALUES (?, ?, ?, ?, ?);

SELECT id, version, metadata, released_by,
     released_at, is_deprecated
FROM package_versions
WHERE package_id = ?
ORDER BY released_at DESC;

SELECT id, package_id, version, metadata,
     released_by, released_at, is_deprecated
FROM package_versions
WHERE package_id = ? AND version = ?
LIMIT 1;
```

3.6 Artifacts and download counters

Artifacts store metadata that points to externally hosted blobs (e.g., object storage). The download endpoint increments counters and then redirects the client to the blob URL.

Listing 22: Artifact create and list queries

```
SELECT id
FROM package_versions
WHERE package_id = ? AND version = ?;

INSERT INTO artifacts (package_version_id, blob_url,
      filename, size_bytes, created_at)
VALUES (?, ?, ?, ?, datetime('now'));
```

```
SELECT id, blob_url, filename, size_bytes,
     created_at
FROM artifacts
WHERE package_version_id = ?;
```

Listing 23: Artifact download lookup + counter increments

```
SELECT a.id, a.package_version_id, a.blob_url, pv.
     package_id
FROM artifacts a
JOIN package_versions pv ON a.package_version_id =
     pv.id
WHERE a.id = ?;

UPDATE package_versions
SET download_count = COALESCE(download_count,0) + 1
WHERE id = ?;

UPDATE packages
SET download_count = COALESCE(download_count,0) + 1
WHERE id = ?;
```

3.7 Votes and comments

Votes are modeled as a *unique* (user, package) pair and are implemented using a single UPSERT statement. Comments are appended and listed in reverse chronological order.

Listing 24: Vote upsert query (one vote per user per package)

```
INSERT INTO votes (user_id, package_id, value,
     created_at)
VALUES (?, ?, ?, datetime('now'))
ON CONFLICT(user_id, package_id)
DO UPDATE SET value = excluded.value,
              created_at = datetime('now');
```

Listing 25: Comment create and list queries

```
INSERT INTO comments (user_id, package_id,
     package_version_id, body, created_at)
VALUES (?, ?, ?, ?, datetime('now'));

SELECT id, user_id, package_version_id, body,
     created_at
FROM comments
WHERE package_id = ?
ORDER BY created_at DESC;
```

3.8 Search queries (FTS5 + fallback)

The search endpoint in `internal/api/handlers_search.go` supports:

- full-text search with SQLite FTS5 (`packages_fts`) when a query string `q` is provided,
- a compatibility fallback using `LIKE` when FTS5 is not available,
- optional filtering by category and bucket using join tables,
- sorting via `sort=most_downloaded` (default), `sort=random`, or creation time.

Listing 26: FTS5 search (q provided)

```
SELECT p.id, p.name, p.description, p.download_count
FROM packages p
JOIN packages_fts f ON p.id = f.rowid
WHERE f MATCH ?
ORDER BY p.download_count DESC
LIMIT ?;
```

Listing 27: LIKE-based fallback search

```
SELECT p.id, p.name, p.description, p.download_count
FROM packages p
WHERE p.name LIKE ? OR p.description LIKE ?
ORDER BY p.download_count DESC
LIMIT ?;
```

Listing 28: Filter packages by category/bucket IDs

```
SELECT id, name, description, download_count
FROM packages
WHERE id IN (SELECT package_id FROM
     package_categories WHERE category_id = ?)
  AND id IN (SELECT package_id FROM package_buckets
     WHERE bucket_id = ?)
ORDER BY download_count DESC
LIMIT ?;
```

3.9 Notes on parameter binding and portability

All SQL statements use parameter binding (placeholders) instead of string concatenation, which prevents SQL injection in handlers.

- In the current development configuration (SQLite), placeholders use `?`.
- In PostgreSQL, placeholders are typically `$1, $2, ...`; therefore queries should be rebound (e.g., using `sqlx.Rebind`) when running against PostgreSQL.

- The search endpoint uses a fixed `LIMIT` (50) to cap load on read-heavy endpoints.

4 Conclusion

`ebuild-repo` demonstrates that a small, focused registry service can provide strong guarantees for package metadata management without requiring heavyweight infrastructure. The project combines a normalized relational schema, explicit migrations, and a resource-oriented REST API to support the core lifecycle operations needed by a build ecosystem: creating packages, publishing versioned manifests, attaching artifacts, searching, and tracking usage.

From an implementation perspective, the system is intentionally structured around clear boundaries: handler functions validate and authorize requests, while the store layer encapsulates persistence and transactional logic. Authentication is implemented with JWTs and strengthened by practical operational safeguards: refresh-token rotation, server-side revocation checks via hashed tokens, and an append-only audit trail for token actions. Together, these choices improve debuggability and incident response while keeping the external contract simple for a CLI client.

The dual-database development/production approach (SQLite for fast iteration, PostgreSQL for deployment) further supports rapid development while retaining a realistic production posture. However, this split also introduces portability considerations (placeholder rebinding, feature differences such as FTS5 availability) that must be handled carefully as the project evolves.

Future work can improve robustness and usability in several areas: enforcing stricter schema consistency (e.g., nullable foreign keys vs. `ON DELETE SET NULL`), expanding PostgreSQL-specific implementations for search and indexing, adding automated synchronization for the FTS content table, and tightening scope representation (e.g., normalizing scopes/allowed package lists instead of storing them as text blobs). In addition, more comprehensive tests, rate limiting, and structured observability (metrics and tracing) would strengthen the service for real-world usage.

Overall, the project provides a practical foundation for a versioned package metadata registry tailored to an ebuild-based build system, with an emphasis on data integrity, auditability, and predictable API-driven workflows.

A EER Diagram

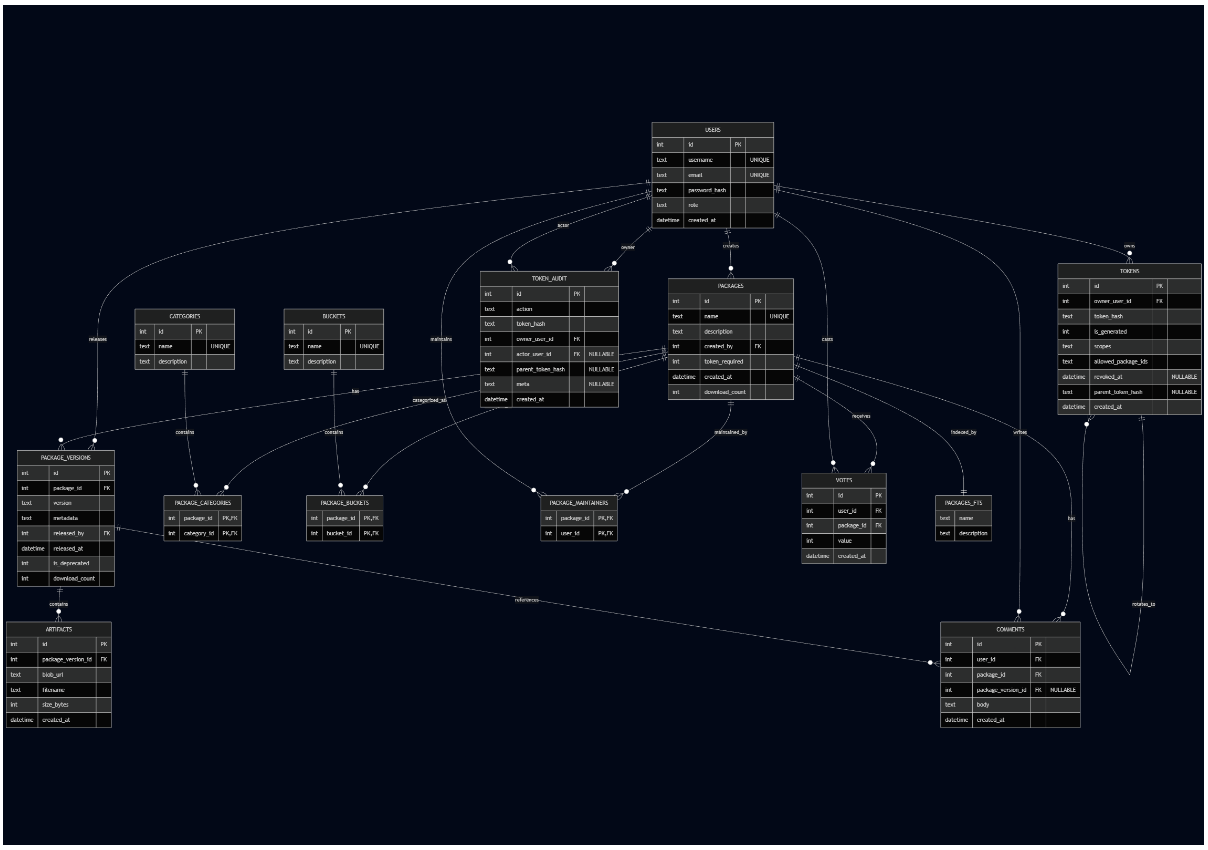

Figure 1: EER diagram of the ebuild-repo relational schema (core tables and relationships).

References

[1] Google. *Documentation - The Go Programming Language.* Jan. 17, 2026. URL: `https://go.dev/doc/`.

[2] PostgreSQL. *PostgreSQL: The world's most advanced open source relational database.* Jan. 17, 2026. URL: `https://www.postgresql.org/`.

[3] pressly. *pressly/goose: A database migration tool. Supports SQL migrations and Go functions.* Jan. 17, 2026. URL: `https://github.com/pressly/goose`.

[4] SQLite. *SQLite Home Page.* Jan. 17, 2026. URL: `https://sqlite.org/index.html`.