The artifact I chose for this database enhancement is my Go-based reverse proxy, which I originally built with an in-memory service registry. While the in-memory approach worked perfectly for development and testing, it had fundamental limitations that made it unsuitable for production environments. Services would disappear on restart, there was no persistence across deployments, and scaling beyond a single instance was impossible. This milestone focused on replacing the ephemeral in-memory registry with a robust PostgreSQL-backed system that provides true persistence, transactional consistency, and the foundation for distributed deployments.

The original registry lived entirely in registry.go as a simple Go map protected by read-write mutexes. It stored service metadata such as names, base URLs, and routing prefixes in memory, which meant that every application restart wiped the slate clean. The test servers would re-register themselves on startup, but any manually registered services or configuration changes were lost. There was no audit trail, no backup capability, and no way to share registry state between multiple proxy instances. While the in-memory approach delivered excellent performance with O(1) lookups, it was fundamentally unsuitable for any serious deployment scenario.

The enhanced version replaces this ephemeral storage with a complete PostgreSQL integration using industry-standard tools: Goose for database migrations and SQLC for type-safe query generation. This transformation required building an entirely new data persistence layer while maintaining full backward compatibility with the existing application interface. The new PostgreSQLRegistry in postgres_registry.go implements the same interface as the original

registry, ensuring that the rest of the applications routing logic, health monitoring, API endpoints, continues to work exactly as before, just with persistent storage underneath.

Implementing the PostgreSQL registry meant solving several technical challenges. First was designing a proper database schema that could handle the array of routing prefixes efficiently. PostgreSQL's native array support allowed storing multiple prefixes as a single TEXT[] column rather than requiring a separate junction table, which simplified queries and improved performance. The migration in 001_create_services.sql creates the services table with appropriate indexes and constraints, establishing a solid foundation for the registry data.

The SQLC integration was particularly important for maintaining code quality and type safety. Rather than writing raw SQL strings scattered throughout the code, I defined all database operations in services.sql using SQLC's annotation syntax. This generates fully type-safe Go functions that handle parameter binding, result scanning, and error handling automatically. The generated code in db provides methods like RegisterService, GetAllServices, and DeleteService that integrate seamlessly with the PostgreSQL driver while eliminating entire classes of SQL injection vulnerabilities and runtime type errors.

The new registry supports upsert semantics through PostgreSQL's ON CONFLICT clause, meaning services can be re-registered with updated configurations without manual deregistration. This is crucial for dynamic environments where services might restart with different port numbers or add new routing prefixes. The database automatically handles timestamp updates, maintains foreign key relationships, and ensures data consistency even under concurrent access from multiple proxy instances.

Maintaining interface compatibility was essential to avoid breaking changes across the application. The RegistryInterface in app.go defines the contract that both registry implementations must follow, including methods for service registration, deregistration, path-based routing, and HTTP endpoint handling. This abstraction allows the application to switch between in-memory and PostgreSQL backends seamlessly, with the main constructor detecting database availability and falling back gracefully if PostgreSQL is unavailable.

The health monitoring system required special attention during this transition. The original health monitor was tightly coupled to the in-memory registry structure, directly accessing the internal map to iterate over services. The enhanced version uses the standardized interface methods, calling GetServers() to retrieve the current service list rather than assuming direct memory access. This change makes the health monitor more robust and allows it to work consistently regardless of the underlying storage mechanism.

Testing the integration revealed the power of the new persistence model. Services registered through the API endpoints are immediately stored in PostgreSQL and survive application restarts. The registry API at /registry returns data directly from the database, and path-based routing queries the persistent storage in real-time. The database serves as the single source of truth for service configuration, eliminating the synchronization issues that would arise in multi-instance deployments.

This enhancement fundamentally transforms the reverse proxy from a development tool into a more production-ready infrastructure. The persistent registry enables horizontal scaling, provides audit trails for service registrations, and creates opportunities for more sophisticated

features like service discovery integration, configuration management, and deployment automation. Someone examining this code can see a clear understanding of database design, transaction management, interface abstraction, and the architectural patterns required for building reliable distributed systems.

The enhancement process reinforced several important lessons about software architecture and database integration. Designing schemas that balance normalization with performance requires careful consideration of query patterns and access frequency. Maintaining backward compatibility while completely reworking underlying storage mechanisms demands thoughtful interface design and comprehensive testing. Integrating code generation tools like SQLC into the development workflow creates a more robust and maintainable codebase, but requires understanding the trade-offs between flexibility and type safety.

The most challenging aspect was ensuring that all the existing functionality—routing, health monitoring, API endpoints—continued to work exactly as before while completely changing the storage backend. This required careful attention to interface design, thorough testing of edge cases, and systematic verification that the enhanced version maintained full behavioral compatibility with the original implementation.

This database integration represents a significant architectural maturity milestone for the reverse proxy. It demonstrates the systems thinking, database design skills, and engineering discipline required to evolve a prototype into production-ready infrastructure that can scale beyond single-instance deployments and provide the reliability guarantees expected in real-world environments.