Repositories in Domain-Driven Design

Bridging Domain and Persistence in Go

Agenda

1. Why Repositories
2. Definition and Placement
3. Repository Styles
4. Aggregate Example
5. DAO vs Repository
6. Design Guidance
7. Testing Strategies
8. Key Takeaways

Why Repositories

Bridging Domain and Data

- Repositories mediate between domain objects and persistence, keeping the domain clean.
- They give the illusion of an in-memory collection while data lives in durable storage.
- Interfaces belong in the domain layer; infrastructure implements them behind the scenes.
- Promote expressive domain code such as `tenant.Activate()` without leaking database concerns.

Definition and Placement

Evans & Vernon on Repositories

- Evans: A Repository mediates between the domain and data-mapping layers, giving the illusion of an in-memory collection.
- Vernon reinforces the concept: expose repositories only for aggregate roots.
- Deliver domain objects in and out; hide records, SQL, and driver specifics.
- Domain layer relies on a stable contract; implementations can change via infrastructure adapters.

Repository Interface in Go

```go
type TenantRepository interface {
    Add(ctx context.Context, tenant *Tenant) error
    ByID(ctx context.Context, id TenantID) (*Tenant, error)
    Remove(ctx context.Context, id TenantID) error
}
```

- Methods reflect aggregate lifecycle operations rather than CRUD verbs alone.
- Keeps invariants within the aggregate while infrastructure handles serialization.

Repository Styles

Collection-Oriented Style

- Mimics an in-memory collection; Unit of Work tracks changes automatically.
- Common in ecosystems with rich ORMs (Java, .NET).
- Works best when the tooling offers change tracking and implicit transactions.
- Less common in Go, but useful to understand when collaborating across stacks.

Persistence-Oriented Style

```go
type TenantRepository interface {
    Save(ctx context.Context, tenant *Tenant) error
    Delete(ctx context.Context, id TenantID) error
    ByID(ctx context.Context, id TenantID) (*Tenant, error)
}
```

- Dominant approach in Go: explicit reads and writes per call.
- No hidden Unit of Workcallers orchestrate state changes intentionally.
- Encourages clarity about I/O and transaction boundaries.

Aggregate Example

Tenant Aggregate Governing Users

```go
type Tenant struct {
    ID       TenantID
    Name     string
    IsActive bool
    Users    []*User
}

func (t *Tenant) Activate() { t.IsActive = true }

func (t *Tenant) RegisterUser(email string) *User {
    user := &User{Email: email, TenantID: t.ID}
    t.Users = append(t.Users, user)
    return user
}
```

- Tenant acts as aggregate root, enforcing invariants for contained users.
- User data reaches persistence only through the tenant repository.

Application Service Coordinates Persistence

```go
func (s *TenantService) ActivateTenant(ctx context.Context, id TenantID) error {
    tenant, err := s.repo.ByID(ctx, id)
    if err != nil {
        return err
    }
    tenant.Activate()
    return s.repo.Save(ctx, tenant)
}
```

- Application layer owns transaction scope and orchestration.
- Repository remains focused on persisting aggregate state.

DAO vs Repository

Comparing DAO and Repository

Aspect	**DAO**	**Repository**
Focus	Tables / records	Aggregates / roots
Returns	DTOs or raw rows	Domain objects
Layer	Infrastructure	Domain interface, infra implementation
Concern	CRUD mechanics	Aggregate lifecycle & invariants

- Both can coexist: DAO encapsulates data access, repository composes domain-facing contract.
- Keeps the domain shielded from persistence technology churn.

Design Guidance

When to Use a Repository

- Provide one repository per aggregate root; avoid exposing internal entities.
- Shape methods around domain language*ActivateTenant*, *RegisterUser*, etc.
- Let repositories persist lifecycles; avoid mixing analytics or logging concerns.
- For cross-aggregate projections, create dedicated read models or query services.

Queries and Consistency

- Scope repository queries to identifiers or local keys within the aggregate boundary.
- Use query services/read models for reporting or multi-aggregate searches.
- Embrace CQRS-style separation when projections must remain eventually consistent.
- Keeps write-model repositories lean and focused on invariants.

Testing Strategies

Testing Repository Contracts

```go
type InMemoryTenantRepo struct {
    store map[TenantID]*Tenant
}

func (r *InMemoryTenantRepo) Save(ctx context.Context, t *Tenant) error {
    cp := *t
    r.store[t.ID] = &cp
    return nil
}

func (r *InMemoryTenantRepo) ByID(ctx context.Context, id TenantID) (*Tenant,
     error) {
    return r.store[id], nil
}
```

- In-memory fakes accelerate unit tests and validate domain logic.
- Integration tests with the real database prove mapping fidelity.
- Watch for aliasing or shared references in fakes to avoid optimistic tests.

Key Takeaways

Recap

- Repositories bridge domain behavior and persistence without leaking infrastructure.
- Prefer explicit, persistence-oriented methods in Go; know when collection-style applies.
- Keep repository interfaces small, aggregate-focused, and transaction-free.
- Separate read models for projections; test with both in-memory and database-backed implementations.
- A well-crafted repository sustains the domain between requestsnot merely a CRUD wrapper.