

• JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#48

CLEAN ARCHITECTURE & THE ART OF MOCKABILITY

STRUCTURING YOUR GO CODE FOR A DECADE OF CHANGE





Protecting the Business Logic

The Core Principle - Dependency Inversion

In traditional architecture, the "Business Logic" depends on the "Database". In Clean Architecture, we flip this. Both depend on an abstraction (an interface).

This is Dependency Inversion.

Your core logic – the code that calculates prices, validates users, or processes orders – should be the "dumbest" part of your system regarding infrastructure. It shouldn't know if it's talking to a SQL database or a CSV file. It just knows it has a Saver interface.





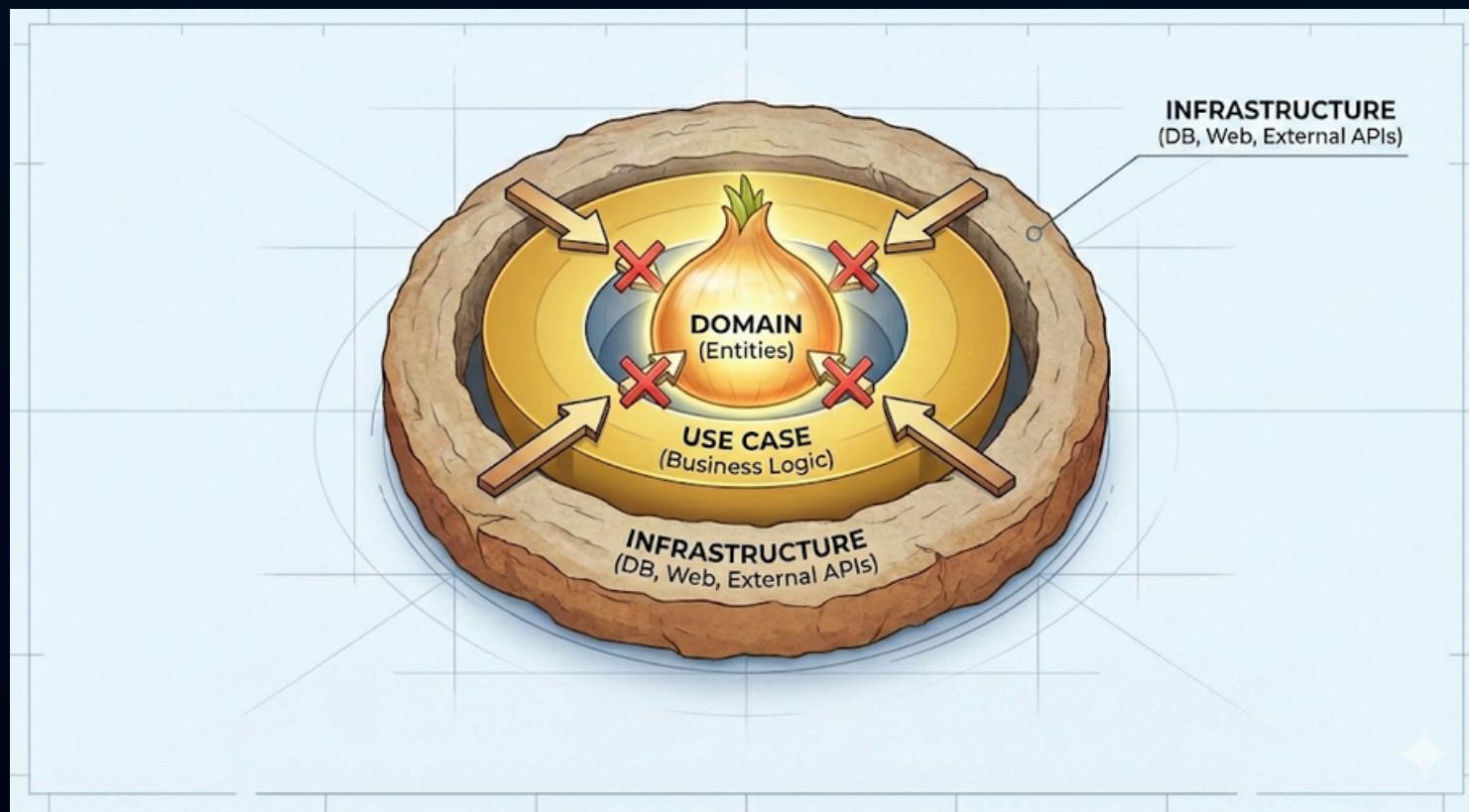
Layered Design: The Onion

The Anatomy of Layers

Think of your app as an onion. The center is the Domain (Entities). Around it is the Use Case (Business Logic). The outermost layer is Infrastructure (DB, Web, External APIs).

The Rule:

Dependencies only point inward. Inner layers never know anything about outer layers. This ensures that the code that earns the company money (the business logic) is never held hostage by the framework you chose to use for your API.





Designing for Testability

Why We Mock

If your business logic is correctly isolated, unit testing it becomes trivial. You don't need a real database or a real internet connection to test an order calculation. You simply provide a "Mock" implementation of your interfaces.

The Strategy:

A mock is a piece of code that mimics a real dependency. It allows you to simulate failures (like a DB timeout) or specific data states (like a user not found) with 100% predictability, making your test suite fast and reliable.





Implementing a Mock

Hand-rolled Mocks

While libraries like *mockery* are popular, hand-rolling a mock in Go is often faster and clearer. It's just a struct that satisfies the interface you defined in your business layer.

```
● ● ●  
  
// The interface in your 'Use Case' layer  
type UserRepository interface {  
    Get(id string) (*User, error)  
}  
  
// The Mock in your 'Testing' file  
type MockUserRepo struct {  
    OnGet func(id string) (*User, error)  
}  
  
func (m *MockUserRepo) Get(id string) (*User, error) {  
    return m.OnGet(id)  
}
```





The Boundary Pattern

Mapping Data Between Layers

A common mistake is passing your "Database Struct" (complete with GORM or JSON tags) all the way into your Business Logic. This leaks infrastructure details into the core. If your database schema changes, your business logic shouldn't have to care.

The Fix:

Keep your layers decoupled by using distinct data models. Use "Mappers" to convert a DBUser into a DomainUser at the boundary. It feels like extra work initially, but it prevents a "Database Schema Change" from forcing a rewrite of your business rules.



```
// Avoid: Passing DB-tagged structs to business logic
// Better: Mapping at the boundary

func (r *userRepository) Get(id string) (*domain.User, error) {
    var dbUser DBModel
    if err := r.db.First(&dbUser, id).Error; err != nil {
        return nil, err
    }

    // Convert infrastructure model to domain entity
    return &domain.User{
        ID:    dbUser.ID,
        Name: dbUser.FullName, // Field name differs from DB
    }, nil
}
```





Interface Segregation

Keep Your Mocks Small

Don't create giant interfaces like *Database* that have 50 methods. If your *OrderService* only needs to *Save* an order, it should only accept an interface that has a *Save* method.

Small interfaces are easier to satisfy, easier to mock, and more reusable. This is the Interface Segregation Principle in action. It allows you to compose complex behavior from tiny, testable building blocks.

```
// Instead of a giant 'DB' interface, use:  
type OrderSaver interface {  
    SaveOrder(ctx context.Context, o *Order) error  
}  
  
func ProcessOrder(o *Order, s OrderSaver) error {  
    // We only need to save, so we only ask for a saver  
    return s.SaveOrder(context.Background(), o)  
}
```





The "Real" Implementation

Plugging in the Infrastructure

Once the business logic is tested and locked in, the "Infrastructure" layer implements the interfaces. This is where you write your SQL queries or call external APIs.

The Integration:

Because the business logic is decoupled, you can delay the decision of which database to use until the very last moment. You can even swap implementations for different environments (e.g., an In-memory implementation for local dev and Postgres for production).





Summary:

- **Business First:** Business logic should never depend on infrastructure.
- **Inward Flow:** Dependencies only point toward the core (Use Cases/Entities).
- **Isolation:** Use different models for the DB and the Domain to prevent leaks.
- **Granularity:** Use small interfaces to make mocking and testing effortless.

Tomorrow we start our dive into observability with Structured Logging (slog) and the "pro" way to trace your app.

