JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#47

DEPENDENCY INJECTION: THE GO WAY

BUILDING FLEXIBLE, TESTABLE SYSTEMS WITHOUT THE "MAGIC" FRAMEWORKS

The Philosophy of Clarity

Explicit over Implicit

Many languages rely on "Magic" DI containers that use reflection to inject dependencies at runtime. Go takes the opposite path. We prefer Explicit Injection: passing exactly what a component needs through its constructor or as a parameter.

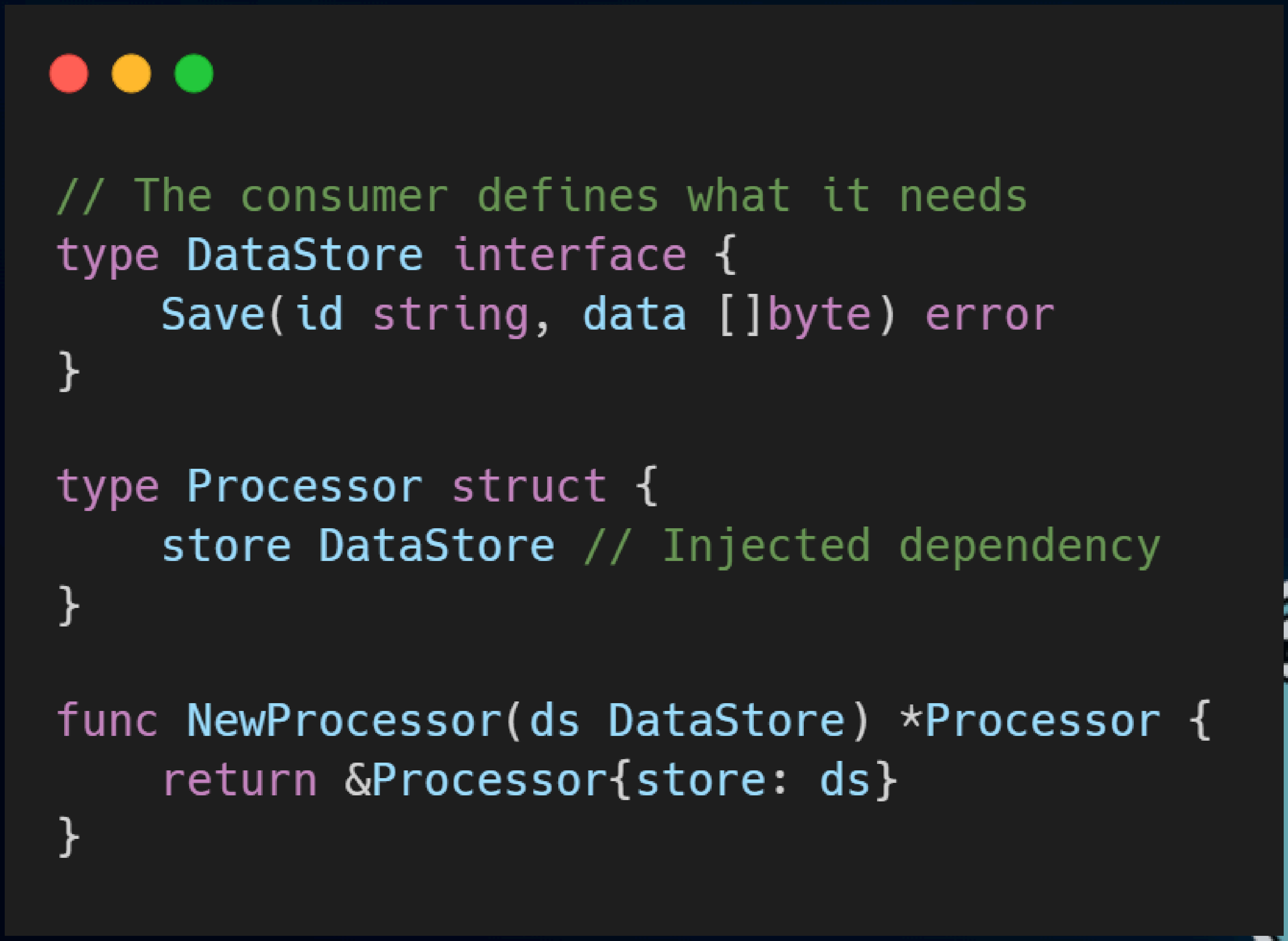
The Reality is when you look at a Go struct, you should know exactly what it depends on just by looking at its fields. No hidden magic, no XML config files, and no "Global Singletons" that make testing a nightmare.

Designing with Interfaces

Accept Interfaces, Return Structs (Remember that??)

The secret to flexible DI in Go is this golden rule. By accepting an interface, your component doesn't care how a task is done, only that the contract is met. By returning a concrete struct, you provide the caller with the most information possible about what they just created.

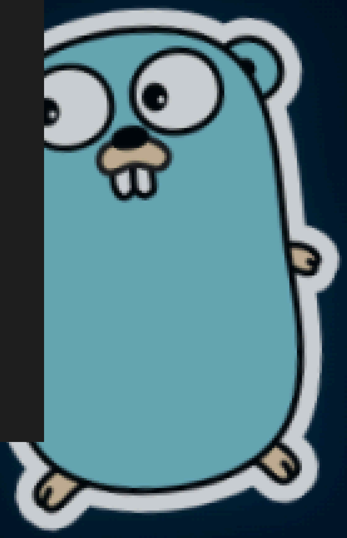
The Pro Move:
Don't define interfaces in the package that implements the logic. Define them in the package that uses the logic. This keeps your packages decoupled and prevents circular dependencies.



```go
// The consumer defines what it needs
type DataStore interface {
    Save(id string, data []byte) error
}

type Processor struct {
    store DataStore // Injected dependency
}

func NewProcessor(ds DataStore) *Processor {
    return &Processor{store: ds}
}
```



Constructor Injection

The "New" Pattern

The most common way to inject dependencies is through a constructor function, typically named *New[Type]*. This ensures that a struct can never be initialized in an "invalid" state - if it needs a database handle to function, the constructor forces you to provide it.

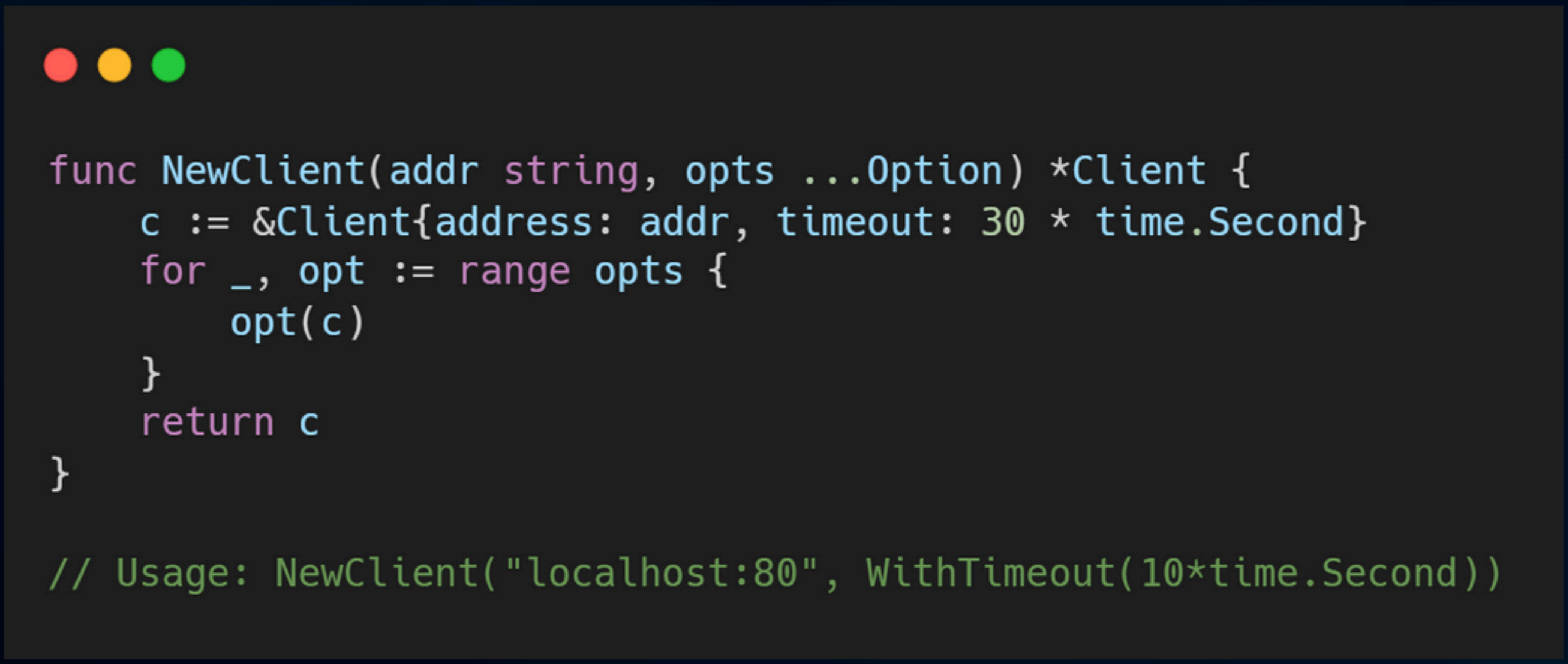
Think of the constructor as the gatekeeper. It ensures that by the time your business logic starts running, all its "tools" are already in its hands and ready to use.

```go
func NewServer(db *sql.DB, logger *slog.Logger) *Server {
    return &Server{
        db:     db,
        logger: logger,
    }
}
```

Handling the "Extra" Stuff

Functional Options for Optional Deps

What if some dependencies are optional? Rather than bloating your constructor with nil pointers, Go uses Functional Options. This pattern allows for a clean, expressive API that scales as your component grows more complex.
This is how high-end Go libraries (like gRPC or the standard *http.Server*) handle configuration. It keeps the "required" dependencies front-and-center while allowing "optional" ones to be tucked away in a readable list of options.



```go
func NewClient(addr string, opts ...Option) *Client {
    c := &Client{address: addr, timeout: 30 * time.Second}
    for _, opt := range opts {
        opt(c)
    }
    return c
}

// Usage: NewClient("localhost:80", WithTimeout(10*time.Second))
```

Avoiding the Global State Trap

The Death of the "*init()*" Singleton

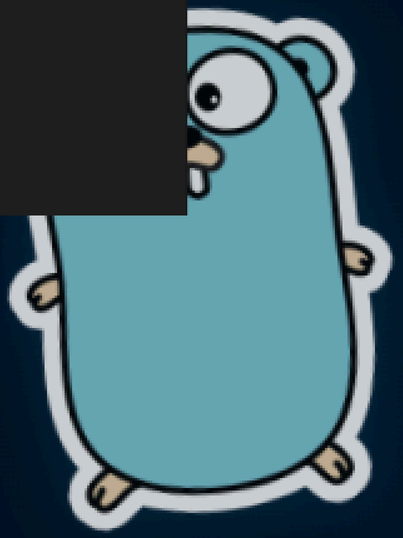
It's tempting to put your Database or Logger in a global variable and initialize it in *func init()*. Resist this urge. Global state makes parallel testing impossible and hides the true dependency graph of your application.

The Architecture:
When you inject dependencies, you can easily swap a real Database for a Mock or a Buffer during a test. If your logic relies on a global *db* variable, you are stuck with a "brittle" test suite that requires a live database to run.

```go
// AVOID: Global variable makes testing & isolation impossible
var db *sql.DB

// BETTER: Pass the dependency to the struct
type Service struct {
    db *sql.DB
}

func (s *Service) GetUser(id string) {
    // Uses s.db instead of global db
}
```

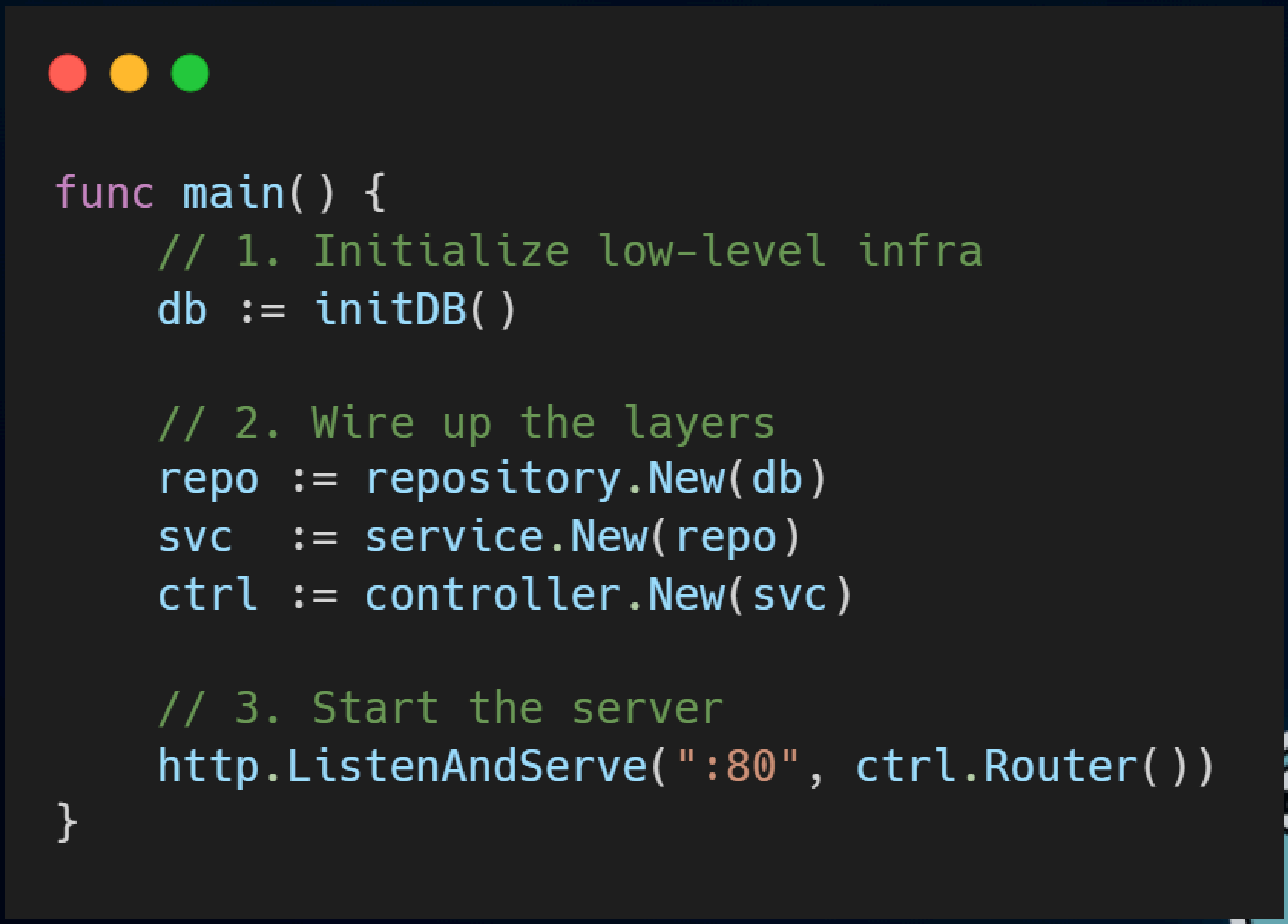


Composing Complexity

Deep Dependency Graphs

In a large application, you might have a *Controller* that needs a *Service*, which needs a *Repository*, which needs a *DB*. This is where manually "wiring" everything in *main.go* becomes the most valuable documentation you have.

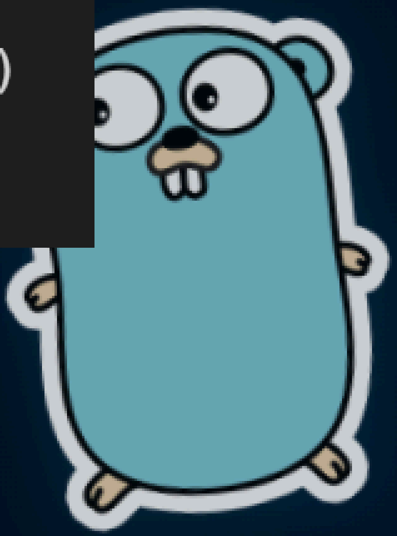
The Big Picture: By wiring your app in *main*, you create a "Source of Truth" for your entire system's architecture. Anyone can look at your main file and see exactly how the data flows from the entry point down to the storage layer.



```go
func main() {
    // 1. Initialize low-level infra
    db := initDB()

    // 2. Wire up the layers
    repo := repository.New(db)
    svc  := service.New(repo)
    ctrl := controller.New(svc)

    // 3. Start the server
    http.ListenAndServe(":80", ctrl.Router())
}
```



When to use a DI Library

Knowing the Limits of "Manual"

Manual DI is the standard for 90% of Go projects. However, if your *main.go* wiring logic starts spanning hundreds of lines, you might look at code-generation tools like Google Wire.

The Distinction: Unlike runtime containers, Wire generates the "manual" code for you at compile-time. You still get the benefits of explicit, compile-safe injection, but without the boilerplate of manually connecting fifty different structs.

Simple Injection, Scalable Systems Recap:

- **Interfaces**: Define them where they are used, not where they are implemented.
- **Constructors**: Use New functions to enforce valid state and required deps.
- **Options**: Use functional options to manage optional configuration cleanly.
- **Main Wiring**: Keep your dependency graph explicit in *main.go*.

Question: Do you find manual wiring in main.go tedious, or do you enjoy having a single place where the entire system's "wiring" is visible? 👇