JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#12

INTERFACES & THE DECOUPLING SECRET

WHY YOU SHOULD ACCEPT INTERFACES AND RETURN CONCRETE STRUCTS

Implicit Satisfaction

No "implements" Keyword

In Go, if a struct has the methods defined by an interface, it is that interface.

```go
type Reader interface {
    Read() string
}

type File struct{}
func (f File) Read() string {
    return "data"
}
// File satisfies Reader automatically
```

This allows you to define interfaces for code you don't even own. You can create an interface in your package to mock a third-party library without the library knowing.

Decouple your Inputs

Accept Interfaces

Your functions should ask for the minimum amount of behavior they need to do their job.

```go
// Bad: Tied to a specific file implementation
func Save(f *os.File){
    // ...
}

// Good: Works with anything that can Write
func Save(w io.Writer){
    // ...
}
```

By accepting an interface, your function becomes instantly testable with mocks and compatible with any future implementation.

Don't Hide the Implementation

Return Structs

Functions should generally return concrete types, not interfaces. This follows the principle: "Be conservative in what you do, be liberal in what you accept."

```go
// The "Java" Way: Returns an interface
// The caller can ONLY see 'Read()'.
// They cannot access 'Close()' or 'Name' without a type assertion.
func OpenInterface(path string) io.Reader {
    return &File{path: path}
}

// The "Go" Way: Returns the concrete struct
// The caller gets the full power of the File struct.
// They can still pass it into any function accepting io.Reader!
func OpenStruct(path string) *File {
    return &File{path: path}
}
```

- If you return an interface, the caller is "trapped" by the methods you chose. If you return a struct, they can satisfy any interface that struct happens to match.
- It’s easier to navigate the codebase. When a function returns *UserStore, I know exactly where the logic lives.
- Returning concrete types helps the compiler's Escape Analysis, often keeping the data on the stack and reducing GC overhead.

Interface Pollution

Stop Creating One-Method Interfaces.

A common junior mistake is creating an interface for every single struct "just in case."

Don't design with interfaces; discover them. Wait until you have at least two or three types that share behavior before abstracting.

*"The bigger the interface, the weaker the abstraction" - Rob Pike.

The Empty Interface

The "any" Type

interface{} (now aliased as *any*) represents a value that could be anything.

The Danger: Using *any* bypasses Go's type safety.
The Senior Take: Use *any* only when you truly don't know the type (like *json.Unmarshal*). If you find yourself using it everywhere, you are likely fighting the language's type system.

Type Assertions & Switches

Recovering the Concrete Type

```go
var i any = "hello"

// Type Switch
switch v := i.(type) {
case int:
    fmt.Println("Integer:", v)
case string:
    fmt.Println("String:", v)
}
```

The Rule: Always use the "comma ok" idiom for assertions (*s, ok := i.(string)*) to avoid runtime panics.

The Method Set Trap

Pointer Receivers & Interface Safety

One of the most common Go compilation errors: "Struct does not implement Interface (method has pointer receiver)." Understanding Method Sets is key to mastering Go's type system.

```go
type Shaper interface {
    Area() float64
}

type Square struct { Side float64 }

// Pointer Receiver
func (s *Square) Area() float64 { return s.Side * s.Side }

// FAIL: A value does not have the pointer method
var _ Shaper = Square{5}

// SUCCESS: The pointer to the struct satisfies it
var _ Shaper = &Square{5}
```

The "Compliance" Trick: How do you ensure your code doesn't break during a massive refactor? Use a Static Interface Check. It costs nothing at runtime but saves you at compile time.

It tells the compiler to try and assign a *nil* pointer of type Square to the Shaper interface, and if you accidentally change the *Area()* method name or signature 6 months from now, your code won't compile, alerting you to the breaking change immediately before you even run a test.

```go
// Place this at the top of your file:
var _ Shaper = (*Square)(nil)
```

Recap:

Design for Decoupling.

- Accept interfaces to make your code testable.
- Return structs to keep your API flexible.
- Use the Compliance Check trick for safety.

Do you find yourself creating interfaces before you have a second implementation, or do you wait for the "Rule of Three"?