● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#30

CUSTOM MARSHALING & OMITEMPTY

WHEN "ZERO" MEANS "MISSING"

The Zero-Value Collision

The omitempty Zero-Value Problem

The *omitempty* tag hides a field if it holds the Go zero-value ("", *0*, *false*, *nil*).

The Bug:
What if *0* is a valid input (e.g., a bank balance or a temperature)?

```go
type Sensor struct {
    Temperature int `json:"temp,omitempty"`
}

// If Temp is 0, the JSON will be: {}
// The consumer won't know if it's 0 degrees or a missing reading.
s := Sensor{Temperature: 0}
data, _ := json.Marshal(s)
```

The Pointer Fix

Distinguishing "Zero" from "Null"

To differentiate between a field being "unset" and a field holding a "zero-value," use a Pointer.

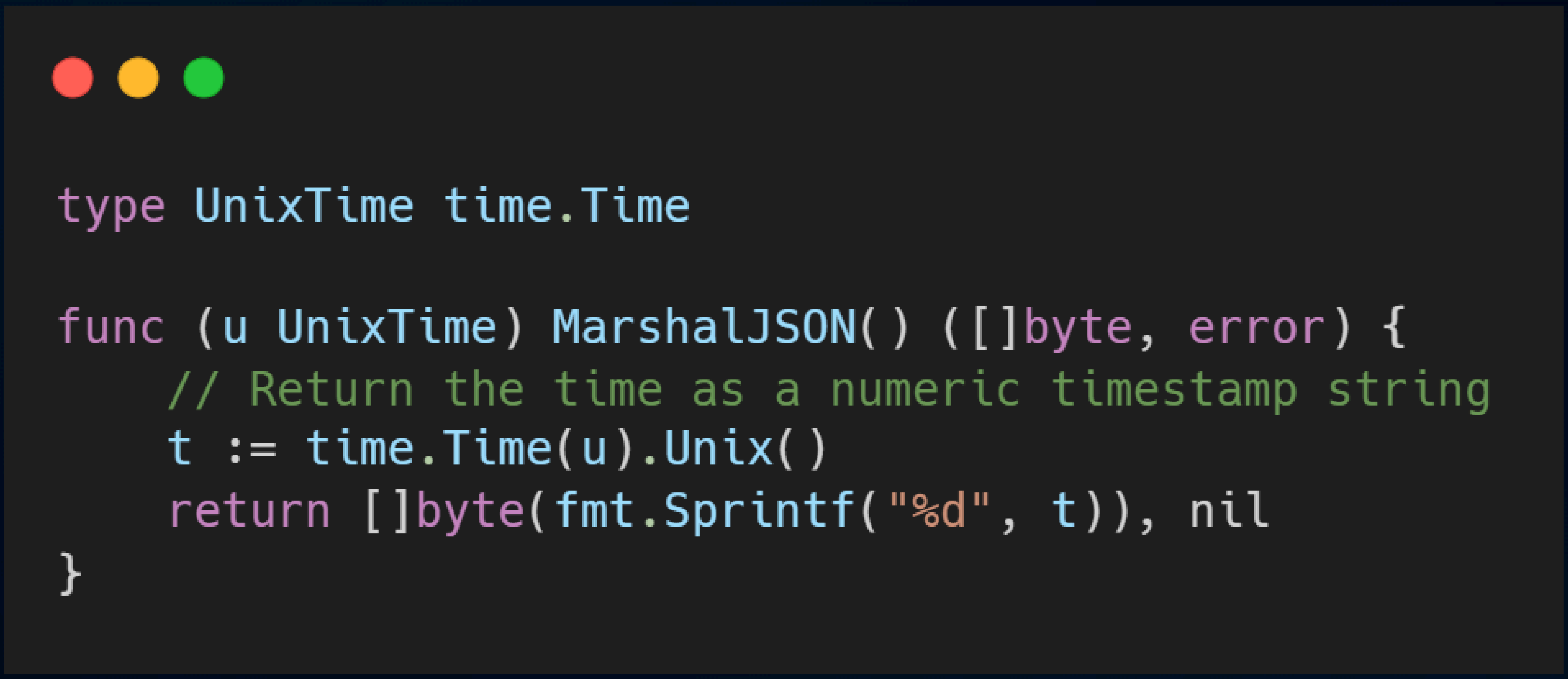
The Logic:
A pointer's zero-value is *nil*. If the pointer is non-nil, even if it points to *0*, *omitempty* will include it.

```go
type Sensor struct {
    Temperature *int `json:"temp,omitempty"`
}

temp := 0
s := Sensor{Temperature: &temp}
// Output: {"temp": 0} -> Correct!

s2 := Sensor{Temperature: nil}
// Output: {} -> Correct!
```

Custom Marshaler Interface

Taking Total Control

Sometimes you need to change the format of a field entirely (e.g., converting a *time.Time* to a Unix timestamp). You do this by implementing *json.Marshaler*.



```go
type UnixTime time.Time

func (u UnixTime) MarshalJSON() ([]byte, error) {
    // Return the time as a numeric timestamp string
    t := time.Time(u).Unix()
    return []byte(fmt.Sprintf("%d", t)), nil
}
```

The Unmarshaler Interface

Parsing Non-Standard Data

Just as you can encode, you can decode custom formats by implementing *json.Unmarshaler*.

Tip: This is incredibly useful for handling "Stringified JSON" inside another JSON field or legacy APIs that return "true"/"false" as strings instead of booleans.

```go
func (u *UnixTime) UnmarshalJSON(b []byte) error {
    var timestamp int64
    if err := json.Unmarshal(b, &timestamp); err != nil {
        return err
    }
    *u = UnixTime(time.Unix(timestamp, 0))
    return nil
}
```

Alias Type Pattern

Avoiding Infinite Recursion

If you call *json.Marshal(v)* inside its own *MarshalJSON* method, you'll trigger a stack overflow.

The **Fix**: Use an Alias Type. It copies the data but strips the methods, allowing you to use the standard marshaler for the rest of the fields.

```go
func (u *User) MarshalJSON() ([]byte, error) {
    type Alias User // Inherits fields, but NOT methods
    return json.Marshal(&struct {
        LastSeen int64 `json:"last_seen"`
        *Alias
    }{
        LastSeen: u.LastSeen.Unix(),
        Alias:    (*Alias)(u),
    })
}
```

Dealing with time.Time

The ISO-8601 Standard

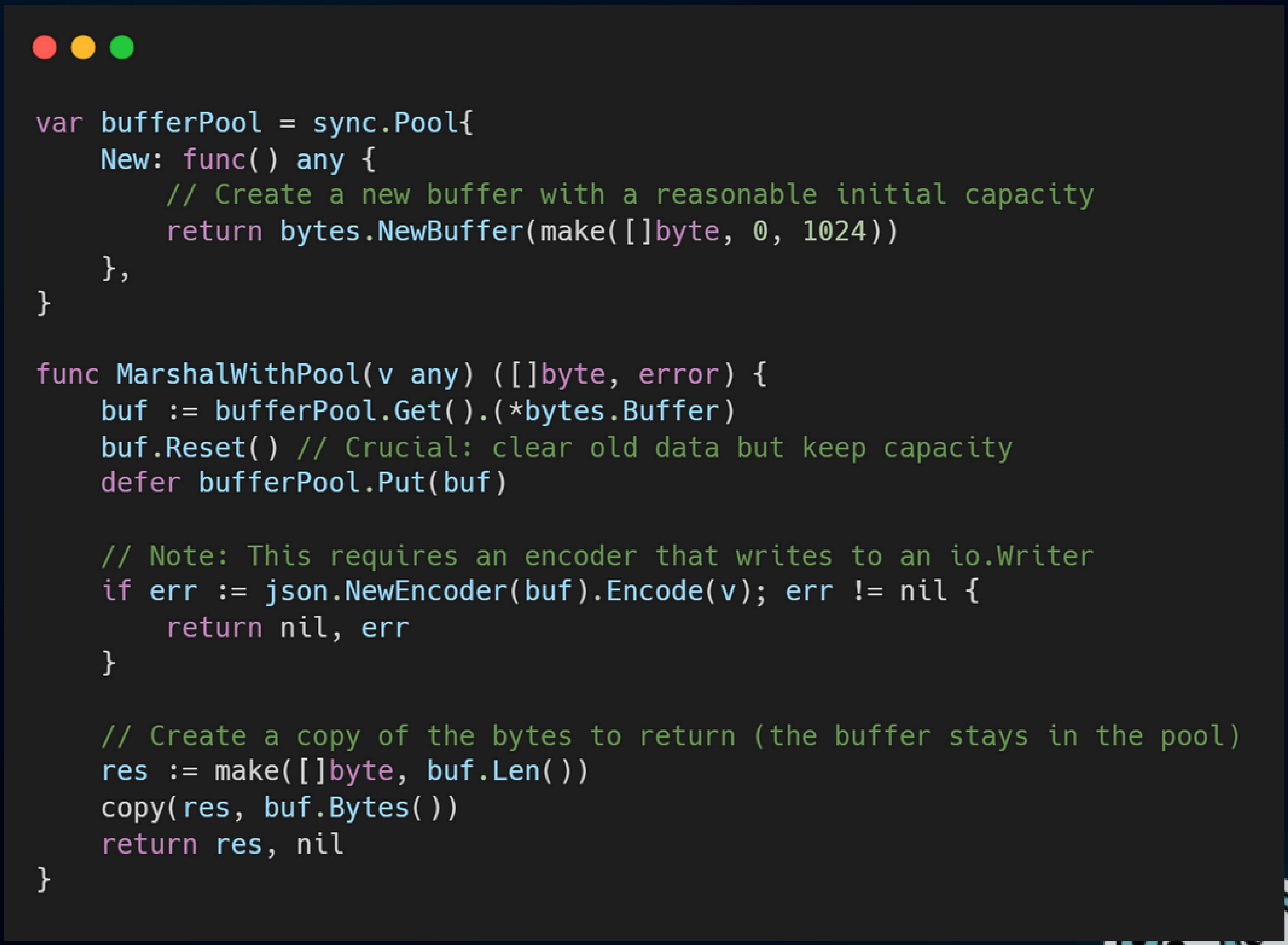
Go's standard library encodes *time.Time* as RFC3339. Most modern APIs expect this, but older systems might struggle with the nanosecond precision Go includes by default.

Golden Rule: Before marshaling, always call *.UTC()* on your time objects. Never send local time offsets to a distributed system; it's the quickest way to create "off-by-one-hour" bugs during Daylight Savings transitions.

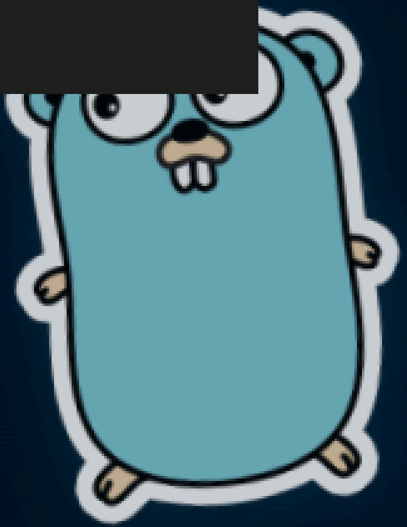
Performance: The Buffer Pool

Senior Optimization: *sync.Pool*

Frequent JSON encoding creates massive pressure on the Garbage Collector (GC) due to short-lived byte slices.



```go
var bufferPool = sync.Pool{
    New: func() any {
        // Create a new buffer with a reasonable initial capacity
        return bytes.NewBuffer(make([]byte, 0, 1024))
    },
}

func MarshalWithPool(v any) ([]byte, error) {
    buf := bufferPool.Get().(*bytes.Buffer)
    buf.Reset() // Crucial: clear old data but keep capacity
    defer bufferPool.Put(buf)

    // Note: This requires an encoder that writes to an io.Writer
    if err := json.NewEncoder(buf).Encode(v); err != nil {
        return nil, err
    }

    // Create a copy of the bytes to return (the buffer stays in the pool)
    res := make([]byte, buf.Len())
    copy(res, buf.Bytes())
    return res, nil
}
```



The Rule:
If your profiling (*go tool pprof*) shows *runtime.mallocgc* or *json.encodeState* as a top CPU consumer, it's time to implement pooling.

Don’t (always) trust the defaults :)

Recap:

- Use pointers for omitempty when 0 or false are valid values.
- Use Alias types in custom marshalers to avoid recursion.
- Always send UTC time in JSON.
- Consider sync.Pool for high-frequency encoding.

Tomorrow we get started with one of Go’s (and my) main strengths - concurrency, goroutines, and much more :)