● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#29

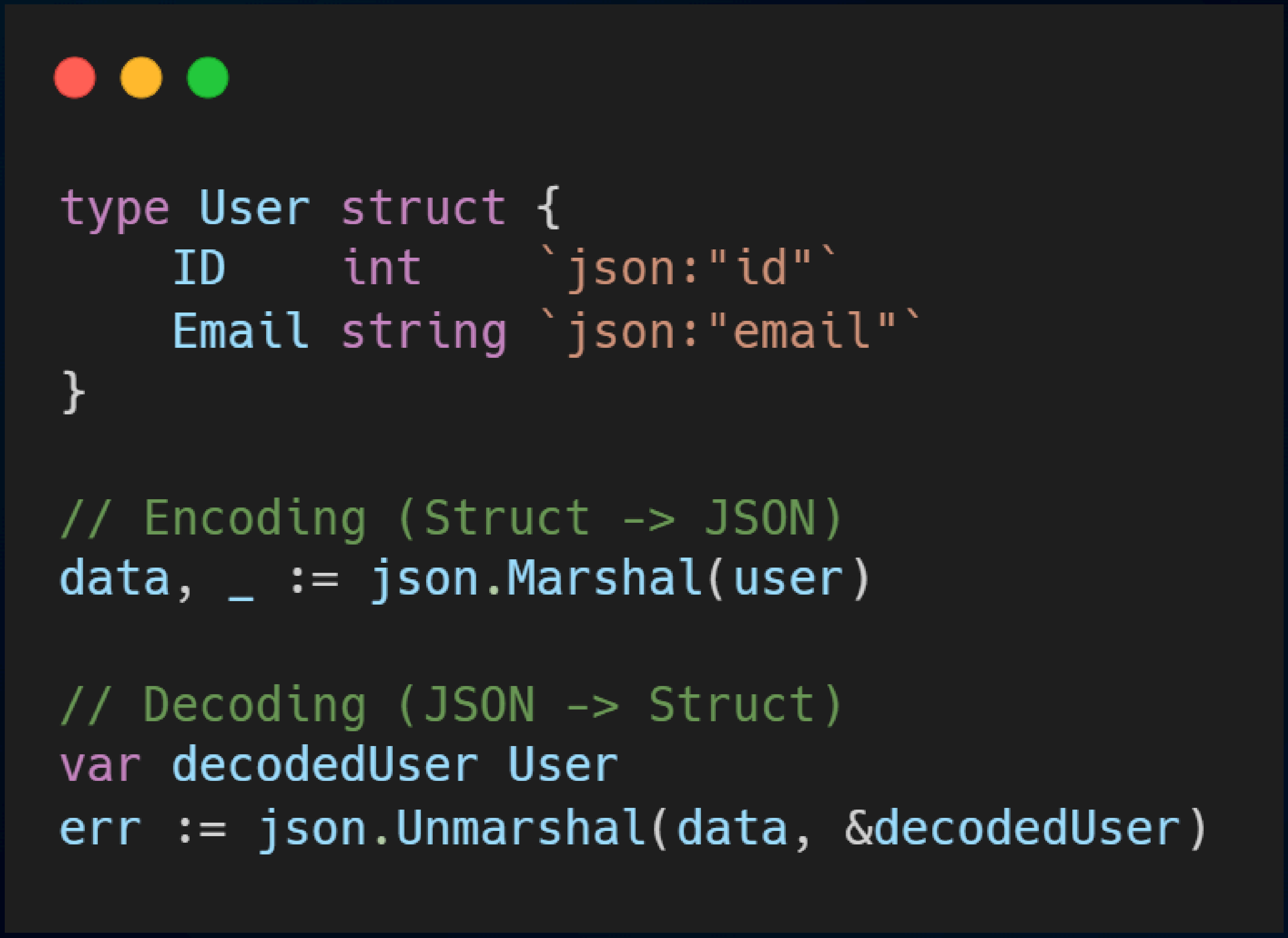
JSON: ENCODING & DECODING

STANDARD LIBRARY VS. PERFORMANCE

Marshal vs. Unmarshal

The Basics (and the Cost)

The *encoding/json* package uses Reflection to map JSON keys to struct fields. This is convenient but expensive in terms of CPU.



```go
type User struct {
    ID    int    `json:"id"`
    Email string `json:"email"`
}

// Encoding (Struct -> JSON)
data, _ := json.Marshal(user)

// Decoding (JSON -> Struct)
var decodedUser User
err := json.Unmarshal(data, &decodedUser)
```

Always use struct tags. Relying on "key matching" (where Go tries to match *Email* to *email*) is slower because Go has to perform case-insensitive string comparisons during the reflection phase.

Stream-Based Processing

Decoder vs. Unmarshal

json.Unmarshal takes a *[]byte*. If you are reading a 100MB JSON file from an HTTP request, you have to load the entire thing into memory first.

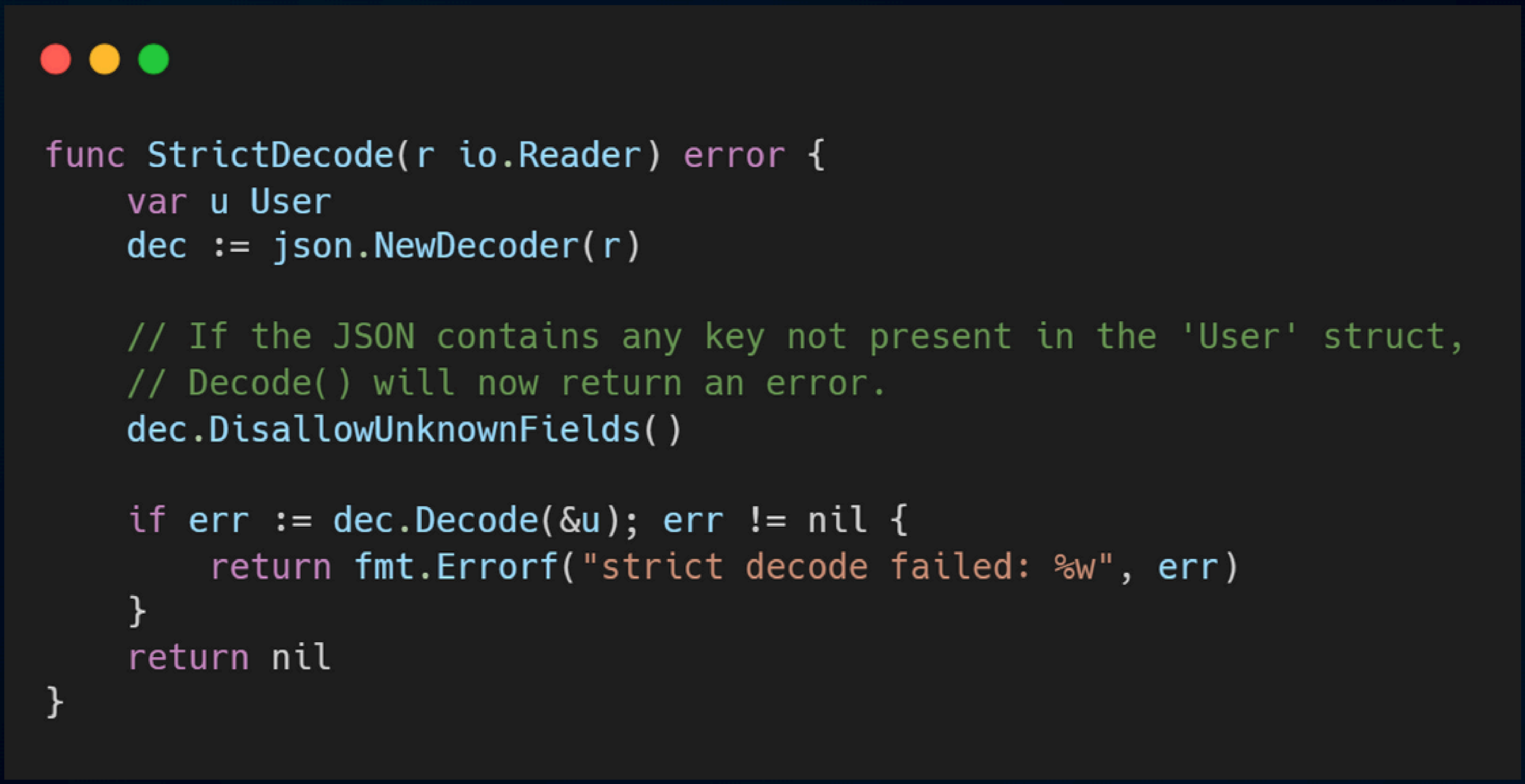
Use *json.NewDecoder*. It reads from an *io.Reader* (like the HTTP request body) and decodes on the fly, keeping your memory footprint low.

```go
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    var u User
    // Decodes directly from the stream. No []byte buffer needed.
    if err := json.NewDecoder(r.Body).Decode(&u); err != nil {
        http.Error(w, err.Error(), 400)
        return
    }
}
```

Handling Unknown Fields

Strict Decoding

By default, *Unmarshal* ignores fields in the JSON that aren't in your struct. In a strict API environment, you might want to reject "noisy" requests.



```go
func StrictDecode(r io.Reader) error {
    var u User
    dec := json.NewDecoder(r)

    // If the JSON contains any key not present in the 'User' struct,
    // Decode() will now return an error.
    dec.DisallowUnknownFields()

    if err := dec.Decode(&u); err != nil {
        return fmt.Errorf("strict decode failed: %w", err)
    }
    return nil
}
```

Strict decoding is your first line of defense against "Mass Assignment" vulnerabilities. It ensures that internal-only fields (like *IsAdmin*) aren't accidentally overwritten by a malicious JSON payload just because you forgot to omit them from a public-facing struct.

Delayed Decoding

The *json.RawMessage* Trick

Sometimes you don't know the schema of a nested object until you read a "type" field. Use *json.RawMessage* to delay the expensive decoding work.

```go
type Envelope struct {
    Type    string          `json:"type"`
    Payload json.RawMessage `json:"payload"` // Stays as []byte
}

// You can now unmarshal Payload into a specific struct
// based on the value of the Type string.
```

This is the standard pattern for Event-Driven architectures or WebSockets where one "Envelope" carries different data shapes.

Dealing with Large Numbers

The Float64 Trap

When decoding into an *interface{}* or *map[string]any*, Go treats all numbers as *float64*. This can lead to precision loss for large 64-bit integers (like Snowflake IDs).

The Fix:
Use *dec.UseNumber()*. It decodes numbers into a *json.Number* type (a string wrapper) which you can then convert to *int64* or *float64* explicitly.

Map vs. Struct

Choosing Your Data Structure

The Trade-off:

- **Structs**: Faster, type-safe, and use less memory. Use these 99% of the time.
- **Maps** (*map[string]any*): Flexible but slow and memory-intensive. Use only when the schema is truly dynamic or unknown.

Do your job:

If you are using *map[string]any* because you are "lazy" to write the struct, you are trading maintainability and performance for 30 seconds of saved typing.

The High-Performance Horizon

When to Leave the Standard Library

If your service spends >30% of its CPU time in *json.Marshal*, the standard library might be your bottleneck.

The Alternatives:

- **easyjson**: Uses code generation to avoid reflection (very fast).
- **jsoniter**: High-performance drop-in replacement.
- **segmentio/encoding/json**: Optimized for modern CPUs.

Senior Take:

Don't switch libraries on day one. Start with the standard library, profile your app (we’ll cover profiling near the end of the series, relax), and only optimize when you have the data to prove it's necessary.

JSON: (Might Be) The Silent CPU Killer.

Recap:

- *Prefer Decoder for streams/requests.*
- *Use json.RawMessage for polymorphic payloads.*
- *Always use struct tags for clarity and speed.*
- *Watch out for number precision in any maps.*

This afternoon, we look at the advanced side: Custom Marshaling and the magic (and danger) of omitempty