Golang Development Standards and Guidelines

This document outlines the standard conventions and best practices for developing Go applications within our organisation. These guidelines ensure code consistency, maintainability, scalability, and security across all projects and teams.

📄 Variable Declaration and Definition in Golang

✅ What Is a Variable?

A **variable** is a named storage location that holds a value. In Go, variables must be declared before they are used.

📦 Variable Declaration Methods

1. Using `var` Keyword (Explicit Declaration)

```
var name string
var age int = 25

```

- Can be used outside functions (at package level)
- Default zero values are assigned if no value is provided

Type	Default Value
`int`	`0`
`string`	`""` (empty)
`bool`	`false`
`pointer`	`nil`

---

2. Short Declaration (`:=`)

```
name := "Alice"
count := 42
```

- **Only valid inside functions**
- Type is inferred
- Cannot be used at package scope

---

3. Multiple Declarations

```
var x, y, z int = 1, 2, 3

```

or with short declaration:

```
a, b := "foo", "bar"

```

---

4. Grouped Declaration

```
var (
    userID   int
    username string = "admin"
    active   bool
)

```

Useful for declaring related variables together.

🧠 Variable Definition vs Declaration

Term	Meaning
Declaration	Tells the compiler about the variable name and type
Definition	Assigns a memory location with value (optional in Go)

🚫 Common Mistakes

Mistake	Why It’s Wrong
Using `:=` outside a function	Only allowed inside function scope
Declaring unused variables	Go compiler throws an error

Using variables before initialization	May lead to logic bugs (though Go uses zero-values)

✅ Best Practices

Do	Don’t
Use short declaration for local vars	Don’t declare and leave unused
Use meaningful variable names	Avoid single-letter names outside loops
Group related variables	Don’t declare global variables without need
Rely on type inference when obvious	Avoid unnecessary type duplication

---

🔍 Example

```go
package main

import "fmt"

var globalCount int = 10 // package-level variable

func main() {
    name := "GoLang"         // short declaration
    var version float64 = 1.20 // explicit declaration

    fmt.Println(name, version, globalCount)
}

```

🗂️ File Naming Conventions in Golang

- **Lowercase letters only**
- **Underscores (`_`) to separate words** (no spaces, no hyphens `-`)
- **Ends with `.go` extension**
- Should be **descriptive** of the contents or functionality

✅ **Examples:**

```
main.go
user_handler.go
db_connection.go
utils.go

```

❌ **Avoid:**

```
UserHandler.go        // Don't use camel case
db-connection.go      // Don't use hyphens

```

🔧 Function Naming Conventions in Golang

1. Exported vs Unexported

- **Exported (public)** functions start with a **capital letter**
- **Unexported (private)** functions start with a **lowercase letter**

✅ **Examples:**

```
func CalculateTotal() int {     // Exported: Accessible from other packages
    return 0
}

func calculateDiscount() int {  // Unexported: Internal use
    return 0
}

```

2. Naming Style

- Use **CamelCase**
- Keep names **short but descriptive**
- **Verb** or **verb+noun** for actions

✅ **Examples:**

```
func GetUserByID(id int) (*User, error)
func sendNotification()
func logError()

```

❌ **Avoid:**

```
func get_user_by_id()   // Don't use snake_case
func x()                // Too short, not meaningful
```

📝 Comments in Golang

🔹 What is a comment?

A **comment** is a note written inside code to explain what the code does.

It is **ignored by the Go compiler** but helps **developers understand the code**.

✅ Types of Comments in Go

1. Single-line Comment

Use `//` to write a comment on one line.

```
// This is a single-line comment
fmt.Println("Hello") // This prints Hello

```

---

2. Multi-line Comment

Use `/* */` for long comments that span multiple lines.

```
/*
This is a multi-line comment.
It can go across several lines.
Useful for big explanations.
*/

```

📍 Where to Put Comments

🔸 Above the package name

```
// Package main is the starting point of Go programs
package main

```

🔸 Above functions

```
// greet prints a welcome message
func greet() {
    fmt.Println("Hi")
}

```

🔸 Inside functions (inline)

```
func greet() {
    // This line prints the greeting
    fmt.Println("Hi")
}

```

🔸 **Above struct or variable**

```
// User represents a system user
type User struct {
    Name string
}

```

⚙️ What Is Call by Value and Reference in Go?

✅ 1. Call by Value – Basics

🔸 **What happens?**

- The function gets a **copy** of the variable.
- Changes inside the function **do NOT affect** the original variable.

✨ **Use when:**

- You don't want the function to change your data.
- You're working with small, simple data (like numbers or short strings).

🔍 **Validation example:**

```
func printAge(age int) {
    if age < 0 {
        fmt.Println("❌ Invalid age")
        return
    }
    fmt.Println("✅ Age is", age)
}

```

✅ 2. Call by Reference – Basics

🔸 **What happens?**

- Function gets a **pointer (reference)** to the variable.
- Changes inside the function **DO affect** the original variable.

✨ **Use when:**

- You want the function to **update the value**.

- You're passing **large data** like a big struct, map, or slice.
- You want to avoid **copying**.

🔍 **Validation example:**

```go
func updateName(name *string) {
    if name == nil || *name == "" {
        fmt.Println("❌ Invalid name")
        return
    }
    *name = "Updated Name"
    fmt.Println("✅ Name updated")
}

```

📦 Example: Value vs Reference

```go
func changeByValue(x int) {
    x = 100
}

func changeByReference(x *int) {
    *x = 100
}

func main() {
    a := 10
    changeByValue(a)
    fmt.Println("After value:", a) // ❌ Still 10

    b := 10
    changeByReference(&b)
    fmt.Println("After reference:", b) // ✅ Changed to 100
}

```

Understanding `defer` in Golang: Why, When, and What to Avoid

✅ 1. Why to Use `defer` in Go

`defer` is used to schedule a **function to run later**, **just before the current function ends**.

✅ **Reasons:**

- **Cleanup**: Always run important cleanup actions like closing files or unlocking resources.
- **Readability**: Keeps the code near where the resource is used.
- **Reliability**: Ensures the code runs **even if the function returns early or panics**.

🔧 **Example:**

```go
func readFile() {
    file, err := os.Open("data.txt")
    if err != nil {
        fmt.Println("❌ Error:", err)
        return
    }
    defer file.Close() // always closes the file!

    // do some reading...
}

```

🕐 2. When to Use `defer`

✅ **Use it when you:**

Scenario	Example
Open a file or DB connection	`defer file.Close()`
Lock a mutex	`defer mu.Unlock()`
Create temporary resources	`defer os.Remove(tmpFile)`
Want to log on exit	`defer log.Println("done")`
Handle panics (safe exit)	`defer recoverFromPanic()`

💡 **Real Use:**

```go
func updateUser() {
    db, _ := sql.Open("driver", "dbsource")
    defer db.Close() // Always close DB connection

    // update logic
}

```

❌ 3. What to Avoid with `defer`

⚠️ Avoid This	Why
Using defer inside loops	Uses memory for each call
Deferring high-cost functions	Delayed execution uses RAM
Relying on execution timing	`defer` runs only at the end

🔍 **Example to Avoid:**

```go
func badLoop() {
    for i := 0; i < 1000; i++ {
        defer fmt.Println(i) // ❌ BAD: creates 1000 deferred calls!
    }
}

```

⚠️ **Instead:**

```go
func goodLoop() {
    for i := 0; i < 1000; i++ {
        fmt.Println(i) // ✅ Prints immediately
    }
}

```

✅ Best Practices for Managing Secrets in Go

🔐 What are "Secrets" in Code?

Secrets refer to sensitive information, such as:

- API keys
- Passwords
- Database connection strings
- Cloud access credentials (AWS, GCP, etc.)
- Encryption keys

⚠️ Why Is This Important?

- ❌ **Hardcoding secrets in Go code is dangerous.**
- It can lead to **security leaks**, especially if the code is pushed to GitHub or shared.

---

1. Never hardcode secrets in code

❌ Bad:

```
const dbPassword = "mySuperSecretPassword"

```

✅ Good: Load it from an **env variable** or a **config file**.

---

2. Use Environment Variables

Go has a built-in way to read environment variables using `os.Getenv()`.

```
import "os"

func connectDB() {
    dbPassword := os.Getenv("DB_PASSWORD")
    if dbPassword == "" {
        log.Fatal("Missing DB_PASSWORD")
    }
    // use dbPassword here
}

```

📌 You can set environment variables in your terminal:

```
export DB_PASSWORD=mysecret123

```

Or in `.env` file (with a loader like `godotenv`)

---

3. Use a `.env` File + `godotenv` Package

Install it:

```
go get github.com/joho/godotenv

```

Create `.env` file:

```
DB_USER=admin
DB_PASSWORD=supersecret

```

Load it in Go:

```
import (
    "github.com/joho/godotenv"
    "os"
)

func init() {
    godotenv.Load() // loads .env file
}

func main() {
    user := os.Getenv("DB_USER")
    pass := os.Getenv("DB_PASSWORD")
}

```

4. Avoid pushing secrets to Git

- Add `.env` to `.gitignore`
- Use tools like [git-secrets]() or `truffleHog` to scan your repo

5. Use Secret Managers (For Production)

In real apps, use tools like:

Tool	Purpose
AWS Secrets Manager	Secure cloud secrets
HashiCorp Vault	Advanced secrets and encryption
Google Secret Manager	GCP-based secrets
Azure Key Vault	Azure secrets

You can integrate them via the SDK or REST APIs in Go.

🚫 What to Avoid

❌ Don’t Do This	✅ Instead, Do This
Hardcode secrets	Use `os.Getenv()` or .env files
Push `.env` to Git	Add `.env` to `.gitignore`
Log secrets	Never log passwords/API keys
Use secrets in public repos	Store them in the env or vaults

📄 Logging Standards

✅ Purpose

To provide a clear and standardised way to log application events, especially errors, including key debug-friendly metadata like function name, line number, error codes, and HTTP status.

🧱 Logging Format

Each log entry should contain the following fields:

Field	Type	Description
`level`	string	Log level (`DEBUG` , `INFO` , `WARN` , `ERROR` , `FATAL`)
`function`	string	Full or shortened name of the function where the log was generated
`line`	int	Line number inside the function (uses `runtime.Caller`)
`message`	string	Short human-readable error message (limited to 80 characters)
`error_code`	string	Internal application error code (e.g., `E1001` , `DB2002`)
`http_code`	int	HTTP status code (e.g., `400` , `500`) for API-level errors

---

🧪 Sample Log Output

```
{
  "level": "ERROR",
  "function": "userService.CreateUser",
  "line": 42,
  "message": "Invalid email format provided by user input...",
  "error_code": "E1002",
  "http_code": 400
}

```

🚦 Log Level Definitions

Level	When to Use
`DEBUG`	Detailed dev/debug info (not in production)
`INFO`	Normal operation messages
`WARN`	Something unexpected but not failed
`ERROR`	Functional failure that needs attention
`FATAL`	Irrecoverable error, will exit or panic

✅ When to Use Logging

Situation	Recommended Log Level	Example
API validation failed	`WARN` or `ERROR`	`E1003`, HTTP 400
Database timeout	`ERROR`	`DB5001`, HTTP 500
Startup success	`INFO`	`App started`
Temporary debug value	`DEBUG`	`len(userList) = 0`

❌ What to Avoid

Anti-Pattern	Recommendation
Logging secrets/passwords	🔒 Mask or skip sensitive data
Hardcoded long error messages	✅ Keep messages concise, max 80 chars
Unstructured logs (e.g., plain print)	✅ Use JSON format for consistency
Logging the same error multiple times	✅ Log once at the correct level
No error codes	✅ Add internal error codes for tracing
Overuse in tight loops	⚠️ May affect performance

🚫 Example: Bad Log

```
fmt.Println("Error occurred while saving user to DB with password:", password)

```

✅ Example: Good Log

```
{
  "level": "ERROR",
  "function": "userService.SaveUser",
  "line": 87,
  "message": "Database timeout saving user",
  "error_code": "DB5001",
  "http_code": 500
}

```

📄 Managing MongoDB Sessions in Golang

---

1. How to Use MongoDB Sessions in Go

Set up Mongo Client (Singleton)

- Create a **single global** `mongo.Client` instance using `mongo.Connect()`.
- Reuse this client across your app — it's **thread-safe**.

```
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()
client, err := mongo.Connect(ctx, options.Client().ApplyURI("your_mongo_uri"))
if err != nil {
    log.Fatal(err)
}
// Use client globally, e.g., assign to a package-level var
```

Use `context.Context` with every operation to control timeouts.

```
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()
collection := client.Database("db").Collection("col")
result, err := collection.FindOne(ctx, bson.M{"name": "abc"}).Decode(&doc)
```

---

2. When to Use MongoDB Sessions

Use Case	Should You Use Session?
Simple single-document query	❌ No, just use collection operations with context
Multi-document atomic transaction	✅ Yes, use `StartSession()` and `WithTransaction()`
Causal consistency (ordering)	✅ Yes, to maintain session state
Long-running or complex ops	✅ Use sessions with context & timeouts
One-off reads/writes	❌ No need for explicit sessions

---

3. What to Avoid When Managing MongoDB Sessions

Mistake	Explanation / Impact
Creating a new client per request	Expensive, resource-heavy, slows app
Not closing sessions (`EndSession()`)	Memory leaks and resource exhaustion
Ignoring context timeouts	Operations may hang indefinitely

Forgetting to close cursors	Causes resource leaks on the server
Using sessions for simple queries	Unnecessary complexity and overhead
Forgetting to disconnect the client on shutdown	Possible connection leaks

---

📄 Constants Handling in Golang

✅ What Are Constants?

- Constants are **fixed values** that **cannot be changed** during program execution.
- They improve readability and maintainability by giving meaningful names to fixed values.

🧱 Declaring Constants

Basic syntax:

```
const Pi = 3.14
const Greeting = "Hello, Go!"

```

You can declare multiple constants at once:

```
const (
    StatusOK   = 200
    StatusFail = 500
)

```

✅ Constant Types

- Constants can be **typed** or **untyped**.
- Untyped constants have more flexibility (can be used as different compatible types).

Example:

```
const x = 42    // untyped int
const y int = 42  // typed int

```

✅ Best Practices for Constants

Do’s	Don’ts
Use uppercase or CamelCase for names	Don’t change constant values
Group related constants with `const` block	Don’t use magic numbers directly
Use `iota` for enumerations	Avoid very large const blocks
Comment constants for clarity	Don’t redefine constants repeatedly

---

🛠️ Example: HTTP Status Codes

```
const (
    StatusOK                  = 200
    StatusBadRequest          = 400
    StatusInternalServerError = 500
)

```

🚩 When to Use Constants

- For fixed values like config keys, status codes, error codes
- When values are logically constant and shared across code

📄 Goroutine Handling in Golang

✅ What is a goroutine?

- A **goroutine** is a lightweight thread managed by the Go runtime.
- Allows concurrent execution of functions.
- Created by prefixing a function call with `go`.

🧱 Starting a Goroutine

```
go func() {
    fmt.Println("Hello from a goroutine")
}()

```

Or with named functions:

```go
func sayHello() {
    fmt.Println("Hello")
}

go sayHello()

```

✅ Best Practices for Goroutines

Practice	Explanation
Use **channels** to communicate	Avoid shared memory; use channels instead
Handle **panics** inside goroutines	Prevent silent failures by recovering panics
Avoid leaking goroutines	Ensure they exit or are cancelled properly
Use **context.Context** to manage lifecycle	Pass `ctx` to cancel goroutines gracefully
Limit number of goroutines	Unbounded goroutines may exhaust resources

🛠️ Synchronizing Goroutines

Using `sync.WaitGroup`

```go
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    // work here
}()
wg.Wait()

```

Using Channels for Signaling

```go
done := make(chan struct{})
go func() {
    // work here
    done <- struct{}{}
}()
<-done

```

🚫 Common Risks to Avoid

Risks	Reason / Fix
Forgetting to wait for goroutines	Main exits before goroutine finishes; use WaitGroup
Data race on shared variables	Use synchronization or channels
Ignoring goroutine leaks	Use contexts to cancel long-running goroutines
Panics in goroutines unnoticed	Use recover inside a goroutine

🧵 Go Concurrency Standards

Channels, Maps, and Locks

🧭 Channels

🔹 Purpose

Channels are used for **communication and synchronization** between goroutines. They help enforce **ownership of data** and reduce reliance on shared memory.

🔹 Declaration

```
ch := make(chan int)            // Unbuffered
ch := make(chan int, 100)       // Buffered (size 100)

```

🔹 Send and Receive

```
ch <- value          // Send
val := <-ch          // Receive

```

🔹 Use Select for Multiplexing

```
select {
case val := <-ch1:
    fmt.Println(val)
case <-time.After(2 * time.Second):
    fmt.Println("timeout")
}

```

✅ Channel Best Practices

Do	Why
Use buffered channels for throughput	Reduces blocking in producers
Close the channel only from the sender side	Prevents panic from multiple closures
Use `select` for multi-channel ops	Enables non-blocking and timeouts
Use `<-chan` and `chan<-` for direction	Enforces read/write contracts

🗺️ Maps

🔹 Purpose

Go's built-in maps (`map[K]V`) are efficient for storing key-value data. **However, they are not safe for concurrent access.**

🔹 Declaration

```
m := make(map[string]int)
m["a"] = 1
val := m["a"]
delete(m, "a")

```

🔹 Safe Access with Locks

```
type SafeMap struct {
    mu sync.RWMutex
    data map[string]string
}

func (s *SafeMap) Get(key string) (string, bool) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    val, ok := s.data[key]
    return val, ok
}

func (s *SafeMap) Set(key, val string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.data[key] = val
}
```

🔹 Using `sync.Map` (Thread-Safe)

```
var sm sync.Map
sm.Store("foo", 1)
val, ok := sm.Load("foo")
sm.Delete("foo")

```

✅ Map Best Practices

Do	Why
Use `RWMutex` for concurrent access	Enables multiple readers and one writer
Use `sync.Map` for unknown/dynamic keys	Ideal for caching, plugin systems, one-off reads
Don’t write to map concurrently	Causes runtime panic

🔒 Locks

🔹 Purpose

Locks (`sync.Mutex` and `sync.RWMutex`) ensure **safe access to shared memory** by serializing read and write access.

🔹 Types of Locks

Lock Type	Use Case
`sync.Mutex`	Simple mutual exclusion
`sync.RWMutex`	High-read, low-write scenarios

🔹 Usage Example

```
var mu sync.Mutex

mu.Lock()
counter++
mu.Unlock()

```

With read-write lock:

```go
var rw sync.RWMutex

rw.RLock()
val := sharedData
rw.RUnlock()

rw.Lock()
sharedData = newVal
rw.Unlock()

```

✅ Lock Best Practices

Do	Why
Use `defer unlock()` after lock	Ensures locks are always released
Keep critical sections short	Prevents long blocking on shared resources
Avoid nested locks	Reduces risk of deadlocks
Use `RWMutex` when reads are frequent	Allows parallel reads

🧠 When to Use What

Situation	Recommended Tool
Goroutines need to coordinate	Channels
Broadcasting shutdown signal	`chan struct{}`
Shared map used by multiple goroutines	`RWMutex` or `sync.Map`
Shared counter or config	`sync.Mutex` / atomic
Multi-stage processing (pipeline)	Channels + Goroutines
Read-heavy, write-rare data	`sync.RWMutex`

📄 File Handling in Go

File handling is a core feature in Go, used for reading, writing, creating, updating, or deleting files from the filesystem. Go's standard `os` , `io` , `bufio` , and `ioutil` (deprecated) packages provide complete support.

📦 Key Packages

Package	Purpose
`os`	Create, open, read, write files
`io`	Low-level stream operations
`bufio`	Buffered IO (more efficient)
`io/ioutil`	Deprecated; use `os` + `io` instead

📂 Common Operations

🔹 1. Create a File

```go
file, err := os.Create("example.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()

```

🔹 2. Write to a File

```go
data := []byte("Hello, file!")
err := os.WriteFile("example.txt", data, 0644)
if err != nil {
    log.Fatal(err)
}

```

> ✅ `os.WriteFile` is new in Go 1.16+ and replaces the older `ioutil.WriteFile`.

🔹 3. Read Entire File

```go
data, err := os.ReadFile("example.txt")
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(data))

```

🔹 4. Read File Line by Line (Buffered)

```
file, err := os.Open("example.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()

scanner := bufio.NewScanner(file)
for scanner.Scan() {
    fmt.Println(scanner.Text())
}
if err := scanner.Err(); err != nil {
    log.Fatal(err)
}

```

🔹 5. Append to File

```
f, err := os.OpenFile("example.txt", os.O_APPEND|os.O_WRONLY, 0644)
if err != nil {
    log.Fatal(err)
}
defer f.Close()

if _, err := f.WriteString("Appended content\n"); err != nil {
    log.Fatal(err)
}

```

🔹 6. Delete a File

```
err := os.Remove("example.txt")
if err != nil {
    log.Fatal(err)
}

```

🔹 7. Check if File Exists

```
if _, err := os.Stat("example.txt"); errors.Is(err, os.ErrNotExist) {
    fmt.Println("File does not exist")
}

```

🔹 8. Get File Info

```go
info, err := os.Stat("example.txt")
if err != nil {
    log.Fatal(err)
}
fmt.Println("Name:", info.Name())
fmt.Println("Size:", info.Size())
fmt.Println("Modified:", info.ModTime())

```

✅ Best Practices

Best Practice	Reason
Always `defer file.Close()`	Prevents file descriptor leaks
Use buffered I/O for large files	More efficient memory usage
Check file permissions (chmod, 0644)	Prevents unauthorized access
Handle `os.IsNotExist` and `os.IsPermission`	Better error granularity
Prefer `os.ReadFile` / `WriteFile` over deprecated `ioutil.*`	Cleaner API (Go 1.16+)

⚠️ Common Mistakes

Mistake	Problem
Not closing files	Leads to resource leaks and locked files
Ignoring error returns	Misses issues like permission or disk errors
Using `ioutil` in Go ≥1.16	Deprecated – use `os` and `io` packages instead
Writing to file without checking mode	Can overwrite when append is needed

---

🔒 File Mode Permissions (Unix)

Mode	Meaning
`0644`	Owner can read/write; group/others can read
`0600`	Owner can read/write only
`0755`	Owner read/write/execute; others read/execute
