ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Best Coding Practices to Improve Performance and Readability of Go Applications

Pallavi Priya Patharlagadda

United States of America

ABSTRACT

The Go programming language, often referred to as Golang, is a popular open-source programming language developed by Google. Go has gained significant popularity in recent years for its simplicity, efficiency, and strong support for concurrent programming. Go is a statically typed, compiled language intended for writing code that is clear and easy to maintain. Go blends the ease of use of dynamically typed languages with the performance and security advantages of statically typed languages. Go is a popular choice for developing command-line tools, data processing pipelines, and scalable and high-performance web servers because of its features. Following best practices is crucial to guaranteeing code quality, readability, and project success as Golang programs scale in size and complexity. In this article, we shall explore the significance of Golang best practices, their crucial role in software development, and the plethora of benefits they bring to both developers and projects.

***Corresponding author**

Pallavi Priya Patharlagadda, United States of America.

Received: February 01, 2024; **Accepted:** February 12, 2024; **Published:** February 16, 2024

Introduction

Go is a statically typed, concurrent, and garbage-collected open-source programming language created at Google in 2009. It is designed to be simple, efficient, and easy to learn, making it a popular choice for building scalable network services, web applications, and command-line tools. The semantics of these open-source programming languages are similar to those of C and Java. Go language promises code efficiency that translates into faster software and apps for businesses. Many Companies recognized the need for lean and efficient code and adopted Golang as their programming language.

Problem Statement

Writing effective code is crucial for developers to create programs that perform well. Go provides developers with a unique opportunity to create effective applications because of its simplicity and performance-focused design. It takes effort and knowledge to write safe and clean code for any project. To safeguard applications against security risks and vulnerabilities, secure coding standards are crucial. It is critical to follow best practices to guarantee code quality, readability, and project success as Golang applications grow in size and complexity. Best practices are tried-and-true methods, procedures, and coding standards that have been upheld over time by experienced programmers because of both their learnings and mistakes. They have proven techniques that make it easier to write code that is straightforward, scalable, highly performant, efficient, resilient, and maintainable. Using these techniques would make it easy to read code and reduce the likelihood of bugs. In this article, we will discuss some essential tips for writing clean code in Go that would help in creating high-quality, maintainable code.

Advantages of go language

- **Fast:** Golang is a compiled language. The code written in Golang compiles into bytecode that runs directly on the machine. This is faster than executing the code in interpreted languages. Faster execution times result in high-performance applications that process large volumes of data. The execution speed of Golang meets expectations even at high loads. Go quickly performs complex calculations, which is especially important for products based on AI, in particular, Machine Learning.
- **Easy to Learn:** The fact that Golang is easy to learn is just another reason to utilize it. Software engineers can easily use Go, especially if they already have a strong foundation in C or Java. Programmers would quickly become accustomed to Go's procedural approach, even though the syntax and keywords may differ slightly.
- **Concurrency:** One of the biggest advantages of Go is its high concurrency. The Goroutines are lightweight functions that can be independent and run together. Since Goroutines are non-blocking, they can handle numerous concurrent methods with a memory footprint of just two kB. In theory, users can execute millions of Goroutines without seeing any system crashes or issues. These Goroutines are managed by Go runtime. For messaging between Goroutines, channels are used. This preserves the order of processing requests and prevents blocking. To block the execution of certain functions until the group of goroutines is completed, wait groups are used. WaitGroups provides the synchronization.
- **Efficient Memory Management:** Efficient memory management is important for large-scale applications that deal with large volumes of data. Go runtime provides a default and optimized Garbage collection mechanism. Go`s

automated garbage collector manages memory allocation and deallocation. Like other languages, Garbage Collector doesn't stop the execution of the application while performing Garbage collection. It runs alongside applications like other goroutines. Hence, Developers can focus on the coding and need not think about memory leakage.

Best Practices for Writing Effective go Application

Proper Indentation

Proper Indentation helps in making the code readable. Use spaces or tabs consistently (preferably tabs) and follow the Go standard convention for indentation.

```
package main

import "fmt"

func main() {
   for i := 0; i < 1; i++ {
      fmt.Println("Hello, Gophers !")
   }
}
```

Go provides a tool called Gofmt which formats the Go code. The gofmt command reads the go program, and it will give you the correctly arranged output after indentation and vertical alignment, and it can even reformat the comments.

gofmt <filename> - prints the formatted code.
gofmt -w <filename>- formats the code and updates the file.
gofmt </path/to/ package> - This will format the whole package.

Error Handling

Errors in Go are values. Return errors from the function and avoid silent error handling (e.g., using '_' for unused variables). Ignoring errors can cause unexpected behavior and increase the difficulty of debugging your code. Utilize Go's built-in error-handling system for handling errors.

Below is an example of bad practice.

```
f, err := os.Open("filename.txt")
if err == nil {
   // work on f
}
```

We are just checking for the success case and not specifying what should happen/return in case of failure. This is a bad practice. Instead, we should handle this way.

```
Func
f, err := os.Open("filename.txt")
if err != nil {
  return err
}
defer f.Close()
// work on f
```

We are catching the error and returning the error to the caller. The calling function can then decide how to proceed in the error scenario. Also, when addressing an error, think about providing more details to the caller. If required, choose to add a structure with errors and other details that the caller would be interested in. This would provide valuable information for debugging and troubleshooting even if the problem occurs in production.

Caution: Please make sure not to return or log any Personal Identifiable Information.
Error handling provides the below benefits.

Safety: Error handling makes sure that unforeseen problems don't trigger a panic or sudden program crash.
CLARITY: It helps you find potential issues in the code and make it easier to comprehend by using explicit error handling.

Few more best practices during error handling

- Errors can be wrapped using the %w verb or inserted into a structured error (such as fs.PathError) that implements Unwrap() error. Error chains are created by wrapped errors, where a new entry is added to the front of the chain with each new layer of wrapping. The Unwrap () error method can be used to navigate the error chain.

- Refrain from duplication. It's usually preferable to let the caller handle any errors you return rather than logging them yourself. The caller has the option to log the mistake or possibly use rate-limit logging giving up on the program entirely or trying to recover. Regardless, allowing the caller to take control prevents log spam.

- Use a log. Error carefully. ERROR level logging causes a flush and is more expensive than lower logging levels. Your code's performance may be greatly affected by this. So, It is always a best practice to have notifications at the error level that should be actionable rather than "more serious" than a warning.

- Errors arising from program initialization, like configuration and bad flags, ought to be transmitted to the main, which ought to make a call to log. Exit with an error message outlining the solution. In these circumstances, log. fatal should not be utilized since an actionable message written by humans is likely to be more valuable than a stack trace that points at the check.

- Avoid the temptation to recover panics to prevent crashes, as this may spread a tainted state. Code review and tests should discover bugs such as accessing an element of a slice that is out of bounds. The more you go from the panic, the less information you have about the program's current status which can include holding locks or other resources. Subsequently, the application can create more unforeseen failure modes that could exacerbate the difficulty in diagnosing the issue. Instead, use monitoring tools to identify unexpected failures and prioritize fixing associated issues rather than attempting to address unforeseen panics in code.

Avoid Repetition when Possible

If you want to use structures in multiple places like controllers and models. Create one common file, and there you can create the structures. Below is one example

```
type Example struct {
   Name string
   Signature string
   Err error
}
```

This structure can be used across the packages instead of defining every time.

Follow Naming Convention

Variable names must be brief and unambiguous, stating the value of the variable. Steer clear of names that are very generic or made up of only one letter. Using descriptive yet short variable names is a good general rule of thumb. Below is an example:

```
// bad
var x int

// good
var numStudents int
```

The names of functions and methods should be precise, and descriptive, and explain what the function or method accomplishes. Use verbs to describe the action the function or method performs. For example:

```
// bad
func x() {
}

// good
func calculateAverage(nums []float64) float64 {
}
```

Follow Single Responsibility Principle

Every method or function should have a single responsibility. A function or method becomes more difficult to read and comprehend if it performs several tasks. It also makes testing and maintenance more challenging. For Example:

```
// bad
func calculateAverageAndSum(nums []float64) (float64, float64)
{}

// good
func calculateAverage(nums []float64) float64 {}
func calculateSum(nums []float64) float64 {}
```

Avoid using Global Variables

Reduce the number of global variables you utilize. These can result in unanticipated behavior making debugging difficult, and code reuse can be hampered. Additionally, they may add needless dependencies between various program components and can cause potential race conditions in concurrent programs. Use local variables instead, send data via function arguments, and return values. For example,

```
// bad
var count int

func incrementCount() {
   count++
}// good
func incrementCount(count int) int {
   return count + 1
}
```

Use Interfaces for Abstraction

To write flexible and maintainable Go code, one must have a solid understanding of Go interfaces. Unlike in many other languages, interfaces in Go are types that provide sets of methods, nonetheless, they are implemented implicitly. This indicates that a type can implement an interface without making any explicit declarations by only implementing its methods. This feature encourages "composition over inheritance," a design philosophy that results in code that is more modular and decoupled. Instead of being in the same package as the implementing struct, interfaces should be specified in the package in which they are utilized. Only the package that uses the interface knows the required methods. Take note that you accept interfaces and return structs.

In Go, simplicity and relevance are key factors to consider while creating reusable and clean interfaces. Interfaces should be small and focused, usually specifying one or two methods. Interfaces promote code reusability and enhance the readability of your codebase by creating interfaces that are precisely specified and strongly aligned with certain actions.

Overgeneralizing an interface or designing an interface with an excessive number of methods are two examples of interface abuse. Interfaces that are too generic may result in types having poorly defined responsibilities, which makes the code more difficult to read and update. An interface with an excessive number of methods may result in bloated applications, where types are forced to implement unnecessary methods, which goes against the minimalist interface design idea. Below is an example interface.

```
// Define the Shape interface
type Shape interface {
   Area() float64
}

// Square struct
type Square struct {
   Side  float64
}

// Circle struct
type Circle struct {
   Radius float64
}

// Implement the Area method for Square
func (s Square) Area() float64 {
   return s.Side *s,Side
}

// Implement the Area method for Circle
func (c Circle) Area() float64 {
   return math.Pi * c.Radius * c.Radius
}
```

Use Defer for Resource Cleanup

By using defer, you may postpone a function's execution until the surrounding function has finished. Program panics will result in the execution of defers. It is frequently used for operations such as file closure, mutex unlocking, and resource release. This guarantees that cleaning activities are carried out even when in error situations. Below is an example.

```
// Open the file "sample.txt"
 file, err := os.Open("sample.txt")
 if err != nil {
```

```
 return err // Exit the program on error
}
defer file.Close() // Ensure the file is closed when the function exits

 // Perform some actions on the file
```

Use go Routines for Concurrency

One of Go's most unique characteristics is its concurrency architecture, which is based on goroutines and channels. Goroutines are lightweight, concurrent threads managed by the Go runtime, and channels are the routes through which goroutines interact, synchronizing execution and sharing data. Compared to conventional thread-based concurrency models, this paradigm enables programmers to build concurrent code that is simpler to comprehend and maintain. Use goroutines to improve performance when handling activities that may be completed simultaneously.

The efficient usage of goroutines and channels is a key component in the implementation of typical concurrency patterns in Go. The 'producer-consumer' pattern is a prevalent one, wherein one or more goroutines generate data, while others utilize it. The data flow across go routines is facilitated via channels.

It is essential in concurrent programming to stay away from race situations and deadlocks. Unpredictable outcomes might arise from race situations, which occur when several goroutines access shared resources without proper synchronization. Go's race detector is a great resource for spotting these kinds of problems. To prevent race conditions, provide appropriate synchronization using mutexes and channels. Conversely, deadlocks happen when processes are waiting on one another, which results in a halt. To prevent these problems, goroutine interactions and channel use must be carefully designed. Ensuring that all channels are properly closed helps in avoiding go leaks. Additionally, it is regarded as a good practice to stay away from unbuffered channels when not required. Here is a sample Go program that uses Goroutines and a wait group.

```
func Samplefunction (i int, wg *sync.WaitGroup) {
        // Do some processing
        wg.Done()
}

func main() {
        no := 3
        var wg sync.WaitGroup
        for i := 0; i < no; i++ {
                wg.Add(1)
                go Samplefunction(i, &wg)
        }
        wg.Wait()
        fmt.Println("All go routines finished executing")
}
```

Use Pointers only if Required

GoLang's improved memory allocation makes data passing by value efficient. When dealing with bigger data structures or when in-place adjustments are unavoidable, utilize pointers sparingly to minimize needless copying.

```
type Person struct {
   name string
   age int
}

func main() {
  // instance of the struct Person
  person1 := Person{"Alice", 25}

  // create a struct pointer
  var ptr *Person
  ptr = &person1

  // print struct instance
  SaveToDB(person1)
}
```

Handle Panics with Recover

To cautiously manage panics and avoid application crashes, use the recover function. Panics in Go are a type of unexpected runtime error that might cause your application to crash. However, Go has a capability called recovery to subtly manage panics. Here is an example.

```
func possiblePanic() {
 defer func() {
  if r := recover(); r != nil {
   // Recover from the panic and handle it gracefully
   fmt.Println("Recovered from panic:", r)
  }
 }()

 // Simulate a panic condition
 panic("Oops! Something went wrong.")
}

func main() {

 // Call the risky operation within a function that recovers from
panics
 possiblePanic()
}
```

Add Comments to the Code

To clarify a function's intent, arguments, and return values, add comments to the function. Additionally, describe the package's purpose as comments at the start of your Go files. Single-line comments can be provided using `//` and multi-line comments can be provided using `/* */`. Follow the `godoc` format for commenting. Below is an example.

```
package main

import "fmt"

// This is the main package of our Go program.
// It contains the entry point (main) function.
func main() {
        userName := "Bob"
    greeting := greetUser(userName)
}

/* greetUser greets a user by name.

 Parameters:
   name (string): The name of the user to greet.
 Returns:
   string: The greeting message.
 */
```

```
func greetUser(name string) string {
   return "Hello, " + name + "!"
}
```

Give Composite Literal Precedence over Constructor Functions

Instead of using constructor functions, generate structure instances using composite literals. Composite literals have several benefits like Readability, Concision, and Flexibility over constructor functions. Here is an example.

```
type Person struct {
 FirstName string // First name of the person
 LastName  string // Last name of the person
 Age       int    // Age of the person
}

func main() {
 // Using a composite literal to create a Person instance
 person := Person{
  FirstName: "Alice",   // Initialize the FirstName field
  LastName:  "Bob",    // Initialize the LastName field
  Age:       35,       // Initialize the Age field
 }

 // Printing the person's information
 fmt.Println("Person Details:")
 fmt.Println("First Name:", person.FirstName)
 fmt.Println("Last Name:", person.LastName)
 fmt.Println("Age:", person.Age)
}
```

Use Explicit Return Values Instead of Named Return Values for Readability

Although named return values are frequently used in Go, they may confuse code, particularly in bigger codebases. Hence it is recommended to use Explicit return values. Here is an example.

```
// namedReturn demonstrates named return values.
func namedReturn(x, y int) (result int) {
   result = x + y
   return
}

// explicitReturn demonstrates explicit return values.
func explicitReturn(x, y int) int {
   return x + y
}
```

Utilize go Testing Framework

Creating dependable and maintainable code in Go begins with writing efficient tests. Writing and running tests are made simple by the substantial support for testing provided by the Go standard library. Go's testing philosophy recommends creating tests in test go files in addition to your code. This approach promotes test-driven development (TDD) and makes it simple to maintain tests as the codebase evolves.

Go code benchmarking is a useful technique, particularly for applications that depend on speed. Writing benchmark tests, which quantify the execution time of your code, is supported by Go's integrated testing infrastructure. These benchmarks are very helpful in pinpointing areas of performance congestion and confirming the effects of improvements. They are defined similarly to unit tests but use the Benchmark function prefix and are run using the go test -bench command.

Go testing frameworks and tools go beyond the built-in library. Mocking and asserting may be done with tools like GoMock or Testify, which offer more advanced features for testing intricate scenarios. Furthermore, coverage tools included in the Go toolchain, such as go test-cover, assist in locating codebase sections that are not sufficiently tested and guarantee comprehensive test coverage across your project. Go developers may make sure that their code works well in a variety of contexts and is both functionally correct and efficient by utilizing these tools and techniques.

Performance Optimization using CPU and Memory Profiling

Profiling is calculating how much memory, CPU time, and other resources your application uses to find bottlenecks. Several built-in profiling tools, such as the pprof package, are available in Go and may be used to collect comprehensive performance statistics. Knowing how your application performs in a production-like setting and where improvements will have the most impact is made possible by this data.

Finding the processes that consume most of the CPU time is made easier with the aid of CPU profiling. Conversely, memory profiling assists in identifying regions of inefficient memory utilization. Following the identification of bottlenecks, attention is directed to improving these areas, which may entail modifications to algorithms, the creation of more effective data structures, or concurrency improvements.

Numerous blogs and forums within the Go community offer real-world experiences of how performance problems were found and fixed. These case studies can provide a wealth of information on useful performance adjustments in Go.

Avoid Shadowing of Variables

When a new variable with the same name is declared within a smaller scope, it's known as shadowing of variables and might result in unexpected behavior. Within that scope, it renders the outside variable of the same name invisible. To avoid confusion, do not shadow variables within nested scopes. Below is the example

```
x:= 10
   fmt.Println("Outer x:", x)

   // Enter an inner scope with a new variable 'x' shadowing the
outer 'x'.
   if true {
      x := 5 // Shadowing occurs here
      fmt.Println("Inner x:", x)
   }

   // The outer 'x' remains unchanged and is still accessible.
    fmt.Println("Outer x after inner scope:", x) // Print the outer
'x', which is 10.
```

CONCLUSION

A thorough grasp of GoLang's design principles and best practices is necessary to write effective code. By sticking to idiomatic code patterns and applying benchmarks and profiling tools, developers may ensure that their systems operate effectively and give optimal performance. It is important to maintain a balance between readability and efficiency, as too intricate optimizations may result in less maintainable code. These best practices are

intended to help you write more reliable, manageable, and effective Go code. They range from effective usage of goroutines and appropriate error handling to performance optimization. Keeping code legible, simple, and manageable is the fundamental goal. Go places a strong emphasis on writing readable code that is straightforward and succinct rather than complex. Keeping this in mind can help you make choices that are consistent with Go's design principles [1-30].

References

1. https://go.dev/doc/effective_go
2. https://go.dev/doc/code
3. https://google.github.io/styleguide/go/best-practices.html
4. https://medium.com/@golangda/golang-quick-reference-top-20-best-coding-practices-c0cea6a43f20
5. https://medium.com/thirdfort/go-best-practices-how-to-code-comfortably-60118a27def8
6. https://go.dev/talks/2013/bestpractices.slide#1
7. https://github.com/pthethanh/effective-go
8. https://codefinity.com/blog/Golang-10-Best-Practices
9. https://www.cloudbees.com/blog/best-practices-for-a-new-go-developer#andrew-gerrand
10. https://aptori.dev/blog/go-secure-coding-best-practices
11. https://dave.cheney.net/practical-go/presentations/qcon-china.html
12. https://blog.stackademic.com/best-practices-in-go-golang-writing-clean-efficient-and-maintainable-code-dccf61542b57
13. https://mobidev.biz/blog/golang-app-development-best-practices-case-studies
14. https://hyperskill.org/learn/step/24920
15. https://www.codingexplorations.com/blog/writing-efficient-go-code-best-practices-for-performant-and-idiomatic-programs
16. https://peter.bourgon.org/go-best-practices-2016/
17. https://golang.withcodeexample.com/blog/introduction-to-golang-best-practices/
18. https://gochronicles.com/writing-go-code-like-a-pro/
19. https://www.xenonstack.com/insights/best-practices-of-golang
20. https://dev.to/apssouza22/golang-best-practices-4fnk
21. https://thegodev.com/best-practices-and-conventions/
22. https://golangdocs.com/golang-best-practices
23. https://levelup.gitconnected.com/10-essential-tips-for-writing-clean-code-in-golang-2d78245a6f40
24. https://clouddevs.com/go/deploying-applications/
25. https://tutorialedge.net/golang/go-project-structure-best-practices/
26. https://www.mytectra.com/tutorials/golang/best-practices-and-tips
27. https://www.uptech.team/blog/why-use-golang-for-your-project
28. https://shaharia.com/blog/writing-clean-and-efficient-code-in-go/
29. https://www.linkedin.com/pulse/how-write-high-performance-code-golang-using-go-routines-kevin-zhou/
30. https://careerkarma.com/blog/golang-best-practices/