

Write Clean Code in Go



Sagar Shrestha · [Follow](#)

Published in readytowork, Inc. · 8 min read · Jul 10, 2023



Major concerning thing while writing your code is how clean is your code. Let's see how to write code in best practices while coding in organizational enterprises. The primary aim is to make the code usable, readable, and maintainable.



GoFmt

Starting from Readability, GoFmt is a package that automatically formats Go program. It uses a camel case and manages spacing/ tabs for clean code. It automatically formats the go program and if it's disabled, you can use this command:

```
go fmt path/to/your/file  
gofmt -w yourfile.go //if you are using different tools, it fmt only if there is
```

It doesn't mean you can't use other practices. You can disable this and work with different practices. This package is particularly inconsistent and bad with swagger fmt.

Test-Driven Development

Test Driven Development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach combining programming, unit test creation, and refactoring.

Test-driven development consists of the following cycle:

1. Write (or execute) a test
2. If the test fails, make it pass
3. Refactor your code accordingly
4. Repeat

Naming Convention

The naming convention is a necessity for clean code. A clear name pattern itself is the documentation for a project. Now let's see how can we write clearly.

Comments

While writing comments, you need to verify if the code needs comments. If the coding pattern is complex and if a comment is helpful, comments are necessary to clarify what's going on.

In GoFmt, all public variables and functions should be annotated with comments.

You should make comments short and comprehensive. The reviewer won't read long comments. For instance:

```
// It iterate from 0 to 99
// and call the do()
// for each iteration
for i := 0; i < 100; i++ {
    do(i)
}
```

This is a tutorial comment. It explains what it's doing. The programmer doesn't need how it is doing but why it has been done. So, try to explain why rather than how.

```
// instantiate 10 threads to handle upcoming workload
for i := 0; i < 10; i++ {
    doSomething(i)
}
```

Now, Let's think about code. The comment might actually be unnecessary when you use a variable name that is specific to that piece of code. That's why we need to give meaningful names.

```
for workerID := 0; workerID < 10; workerID++ {
    instantiateThread(workerID)
}
```

Function Naming

The more specific the function, the more general its name. We want to start with a very broad and short function name that describes the general functionality. It's better to start the function name with a verb such that we know the functionality. Try to make them as short as possible as long names are often confusing

```
func BeerService(beerBrands []BeerBrand) []Beer { // moreGeneral Name
    return GetBeerList(beerBrands)
}

func GetBeerList (beerBrands []BeerBrand) []Beer{ // more specific starting with
    var beerList []Beer
    for _, brand := range beerBrands {
```

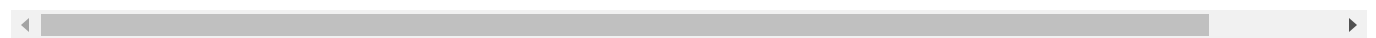
```

        for _, beer := range brand {
            beerList = append(beerList, beer)
        }
    }
    return beerList
}

```

Variable Naming

Our variable names should become less specific as we travel deeper into a function's scope. Better not use a, b, c literal which doesn't explain our functionality.



```

func BeerBrandListToBeerList(beerBrands []BeerBrand) []Beer {
    var beerList []Beer
    for _, brand := range beerBrands {
        for _, beer := range brand {
            beerList = append(beerList, beer)
        }
    }
    return beerList
}
// we can see the variable got less specific as we go down the scope

```

Clean Functions

Try to make smaller functions as possible. It makes our code digestible (understandable). It is not necessarily for avoiding duplication.

It also prevents indentation hell. Try to use a single if- statement and avoid using an else-if statement for better readability. Indentation hell is particularly common when working with `interface{}` and using type casting:

Instead of nesting, we can do something like this

```

func GetItem(extension string) (Item, error) {
    refIface, ok := db.ReferenceCache.Get(extension)
    if !ok {
        return EmptyItem, errors.New("reference not found in cache")
    }

    ref, ok := refIface.(string)
    if !ok {
        // return cast error on reference
    }
}

```

```

    }

    itemIface, ok := db.ItemCache.Get(ref)
    if !ok {
        // return no item found in cache by reference
    }

    item, ok := itemIface.(Item)
    if !ok {
        // return cast error on item interface
    }

    if !item.Active {
        // return no item active
    }

    return Item, nil
}

```

Then make simpler functions like this.

```

func GetItem(extension string) (Item, error) {
    ref, ok := getReference(extension)
    if !ok {
        return EmptyItem, ErrReferenceNotFound
    }
    return getItemByReference(ref)
}

func getReference(extension string) (string, bool) {
    refIface, ok := db.ReferenceCache.Get(extension)
    if !ok {
        return EmptyItem, false
    }
    return refIface.(string), true
}

func getItemByReference(reference string) (Item, error) {
    item, ok := getItemFromCache(reference)
    if !item.Active || !ok {
        return EmptyItem, ErrItemNotFound
    }
    return Item, nil
}

func getItemFromCache(reference string) (Item, bool) {
    if itemIface, ok := db.ItemCache.Get(ref); ok {
        return EmptyItem, false
    }
    return itemIface.(Item), true
}

```

Our main function must be just an abstraction of what needs to be done. The short function doesn't mean you can add everything in one line. It can

sometimes reduce readability.

```
func GetItemIfActive(extension string) (Item, error) {  
    if refIface,ok := db.ReferenceCache.Get(extension); ok {if ref,ok := refIfac  
}
```

Function Signature

Function signatures should only contain one or two input parameters. In certain exceptional cases, three can be acceptable, but this is where we should start considering refactoring. If we need to pass more than that, we need to create a struct and pass it as it is more readable than sending all data at once.

```
type QueueOptions struct {  
    Name string  
    Durable bool  
    DeleteOnExit bool  
    Exclusive bool  
    NoWait bool  
    Arguments []interface{}  
}  
  
q, err := ch.QueueDeclare(QueueOptions{  
    Name: "hello",  
    Durable: false,  
    DeleteOnExit: false,  
    Exclusive: false,  
    NoWait: false,  
    Arguments: nil,  
})
```

This solves these problems: misusing comments, accidentally labeling the variables incorrectly, and mistake ordering. We can also use default values as well.

Variable Scope

One problem with short functions is more global variables. Global variables are problematic and don't belong in clean code. If global variables are mutable, it is hard to trace the content.

Other than avoiding issues with variable scope and mutability, we can also improve readability by declaring variables as close to their usage as possible. Also, it is recommended to declare the return type in the function as figuring out the eventually returned value can be a nightmare. It is better to use it directly after a declaration.

```
package main

import "fmt"

type MyStruct struct {
    ID    int
    Name  string
}

func main() {
    // Declare and initialize variables close to their usage
    count := 0
    message := "Hello, world!"

    // Use variables directly after declaration
    fmt.Println(count)
    fmt.Println(message)

    // Declare return type in function signature
    result := calculateResult()
    fmt.Println(result)

    // Use pointers to limit mutability
    myStruct := &MyStruct{ID: 1, Name: "John"}
    updateName(myStruct)
    fmt.Println(myStruct.Name)
}

func calculateResult() int {
    // Perform some calculations and return the result
    return 42
}

func updateName(s *MyStruct) {
    // Modify the name of the struct
    s.Name = "Jane"
}
```

As there is no way of declaring a `const struct` or `static variable` in Go, it is hard to limit the mutability in Go. This means that we'll have to restrain ourselves from modifying this variable at a later point in the code. You can use pointers to limit some level of immutability.

Returning Defined Errors

In coding, there might be cases when we throw errors directly to illustrate the problem.

```
return Item{}, errors.New("item could not be found in the store")
```

It is harder to track. Instead, we can just store it in a variable, and in later cases, we can use a variable instead of the entire text.

```
package clean
var (
    NullItem = Item{}
    ErrItemNotFound = errors.New("item could not be found in the store")
)

func (store *Store) GetItem(id string) (Item, error) {
    store.mtx.Lock()
    defer store.mtx.Unlock()

    item, ok := store.items[id]
    if !ok {
        return NullItem, ErrItemNotFound
    }
    return item, nil
}

// Now we can use if errors.Is(err, clean.ErrItemNotFound) for error handling
```

If you want to return a specific error message, you can look for returning dynamic errors.

Handling Nil

Things can break when you try to access methods or properties of a `nil` value. Thus, it's recommended to avoid returning a `nil` value when possible.

```
//one way can be justing checking if the data is nil
if app.cache == nil {
    app.cache = NewKVCache()
}
//you can also pass empty struct instead of nil
```

Handling Pointers

The pointer itself is a complex thing. We need to know about itself first to reduce the complexity. First, passing a pointer means the data sent are not deep copies instead, you are passing actual data. Changes in one place can change everywhere else. It seems basic but while handling pointers, this simple thing can corrupt real data. That's why we need to understand the mutability of pointers. Let's look at this code to understand mutability in pointer.

```
type Person struct {
    Name string
}

func changeName(p *Person, newName string) {
    p.Name = newName
}

func main() {
    // Creating a person instance
    person := Person{Name: "John"}
    fmt.Println("Initial name:", person.Name) // Output: Initial name: John
    changeName(&person, "Jane")
    fmt.Println("Updated name:", person.Name) // Output: Updated name: Jane
    // Although name is changed in another function data has been changed in all s
}
```

Interfaces

Without generics, interfaces are the only option to send multitype data. But it is important that we use concrete type whenever possible. Go 1.19 supports generics. Generics can be used to further reduce the redundancy and increase the readability.

Let's look at interfaces method first.

```
type Person struct {
    Name string
}

func changeName(p interface{}, newName string) {
    switch v := p.(type) {
    case *Person:
        v.Name = newName
    case *string:
        *v = newName
    default:
        fmt.Println("Unsupported type")
    }
}
```

```

}

func main() {
    person := Person{Name: "John"}
    changeName(&person, "Jane")

    name := "Alice"
    changeName(&name, "Bob")

    fmt.Println("Updated name:", name) // Output: Updated name: Bob
}

```

if you are looking to send direct data from interface without newName string parameter. You can do something like this

```

func changeName(p interface{}, newName string) {
    if person, ok := p.(*Person); ok {
        person.Name = newName
    } else {
        fmt.Println("Unsupported type")
    }
}

```

But, As I have said earlier, using interface is not concrete. It is better to use concrete example. Since Go 1.19 supports generics, it is better to use generics. Now, Let see how we can implement generics.

```

package main

import (
    "fmt"
    "github.com/cheekybits/genny/generic"
)

type Telephone generic.Type

func GetAllUsers[T Telephone](T Telephone) bool {
    // Logic to get all users based on the provided telephone type
    // Placeholder return statement
    return true
}

func main() {
    telephone := "1234567890"
    result := GetAllUsers(telephone)
    fmt.Println(result)
}

```

This is the end of the article. Let me know what I should do next. Make sure you like and follow for updates. You can pop into my website here:
<http://sagarshrestha1999.com.np/>

Happy Coding 🧑🏻💻

- Clean
- Go
- Practice
- Golang
- Efficient



Published in readytowork, Inc.

Follow

75 Followers · Last published 22 hours ago

Collaborating to incorporate latest trends & technologies in products & services to serve better



Written by Sagar Shrestha

Follow

13 Followers · 9 Following

Software Engineer | Backend Developer | GO | JS | PYTHON
<http://sagarshrestha1999.com.np> <https://www.linkedin.com/in/znerfii/>

No responses yet




What are your thoughts?

Respond

More from Sagar Shrestha and readytowork, Inc.

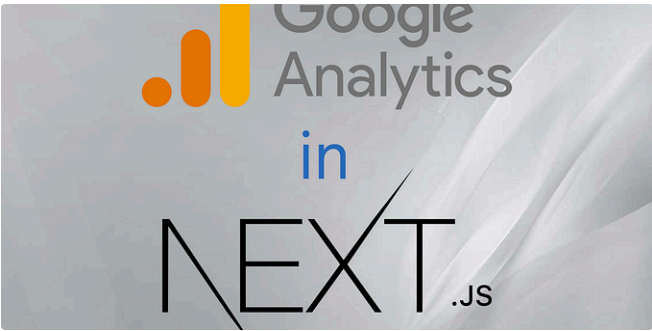


 Sagar Shrestha

Basic CRUD Operation in Ruby on Rails

This tutorial will cover the basic operation we need to know for developing an application i...

Nov 5, 2023  7

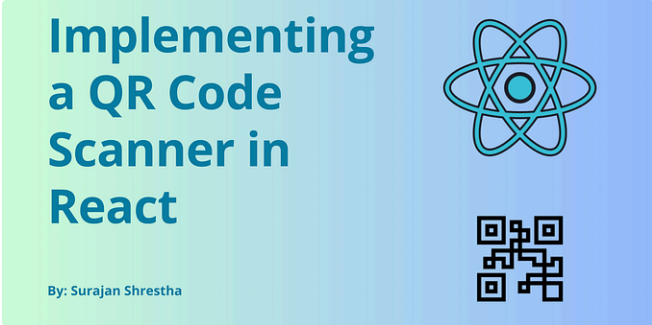



 In readytowork, Inc. by SagarTS

Google Analytics in Next js

Today we will learn on how to add google analytics to our Next js app. Before starting...

Mar 4, 2024  126  7




 In readytowork, Inc. by Surajan Shrestha

Implementing a QR Code Scanner in React

Let's use the qr-scanner package to implement our own lightweight QR Scanner i...

Jan 7, 2024  159  2



 In readytowork, Inc. by Sagar Shrestha

Microservice in Gin Golang

Before we start, this is one of the practice sessions. And you might need a computer...

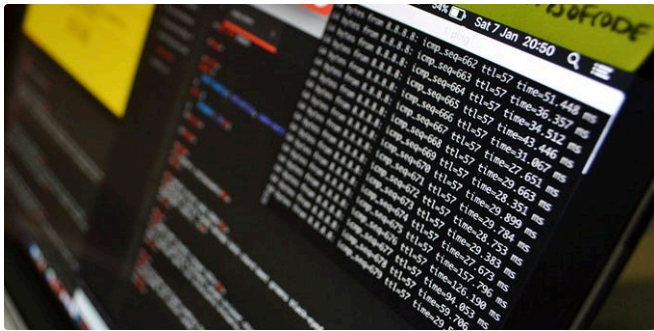
May 2, 2023  11



See all from Sagar Shrestha

See all from readytowork, Inc.

Recommended from Medium



 In Towards Dev by Renaldi Purwanto

10 Hidden Gems in Go (Golang) Every Developer Should Know

Photo by Lewis Kang'ethe Ngugi on Unsplash



 Piotr Persona

Golang Optional type

The following article was inspired by Rust Option type.

Open in app ↗

Sign up

Sign in

Medium



Search



Write

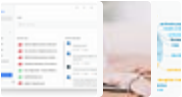


Lists



General Coding Knowledge


20 stories · 1853 saves



Productivity

242 stories · 662 saves



 In Go Golang by Yash

8 Golang Performance Tips I Discovered After Years of Coding

These have saved me a lot of headaches, and I think they'll help you too. Don't forget to...



Oct 17, 2024




816



8



 Golang and Data

Golang Maps

Maps

Jan 1



5





Atharva Pandey

Go Ultimate: The ultimate go cheatsheet you'll ever need

Let's be real — keeping track of syntax and features across multiple programming...



Oct 15, 2024



140



3



In Programmer's Career by Wesley Wei

Goroutine Collaboration You Should Know In Golang

Advanced Techniques for Efficient Concurrency In Go



6d ago



111



See more recommendations