

Not sure how to structure your Go web application?

My new book guides you through the start-to-finish build of a real world web application in Go – covering topics like how to structure your code, manage dependencies, create dynamic database-driven pages, and how to authenticate and authorize users securely.

[Take a look!](#)



Golang Interfaces Explained

Published on: August 6th, 2019

For the past few months I've been running a survey which asks people what they're finding difficult about learning Go. And something that keeps coming up is the concept of *interfaces*.

I get that. Go was the first language I ever used that had

ALEX
EDWARDS

ARTICLES

WORK WITH
ME

ABOUT



2. Explain why they are useful and how you might want to use them in your code;
3. Talk about what `interface{}` (the empty interface) is;
4. And run through some of the helpful interface types that you'll find in the standard library.

What is an interface in Go?

An interface type in Go is kind of like a *definition*. It defines and describes the exact methods that *some other type* must have.

One example of an interface type from the standard library is the `fmt.Stringer` interface, which looks like this:

```
type Stringer interface {  
    String() string  
}
```

We say that something *satisfies this interface* (or *implements this interface*) if it has a method with the exact signature `String() string`.

For example, the following `Book` type satisfies the interface because it has a `String() string` method:

```
type Book struct {  
    Title string  
    Author string  
}  
  
func (b Book) String() string {  
    return fmt.Sprintf("Book: %s - %s", b.Title, b.Author)  
}
```

It's not really important what this `Book` type is or does. The only thing that matters is that it has a method called `String()` which returns a `string` value.

Or, as another example, the following `Count` type *also* satisfies the `fmt.Stringer` interface – again because it has a method with the exact signature `String() string`.

```
type Count int

func (c Count) String() string {
    return strconv.Itoa(int(c))
}
```

The important thing to grasp is that we have two different types, `Book` and `Count`, which do different things. But the thing they have in common is that they both satisfy the `fmt.Stringer` interface.

You can think of this the other way around too. If you know that an object satisfies the `fmt.Stringer` interface, you can rely on it having a method with the exact signature `String() string` that you can call.

Now for the important part.

*Wherever you see declaration in Go (such as a variable, function parameter or struct field) which has an interface type, you can use an object of any type **so long as it satisfies the interface**.*

For example, let's say that you have the following function:

```
func WriteLog(s fmt.Stringer) {
    log.Print(s.String())
}
```

```
}
```

Because this `WriteLog()` function uses the `fmt.Stringer` interface type in its parameter declaration, we can pass in any object that satisfies the `fmt.Stringer` interface. For example, we could pass either of the `Book` and `Count` types that we made earlier to the `WriteLog()` method, and the code would work OK.

Additionally, because the object being passed in satisfies the `fmt.Stringer` interface, we *know* that it has a `String() string` method that the `WriteLog()` function can safely call.

Let's put this together in an example, which gives us a peek into the power of interfaces.

```
package main

import (
    "fmt"
    "strconv"
    "log"
)

// Declare a Book type which satisfies the fmt.Stringer interface.
type Book struct {
    Title string
    Author string
}

func (b Book) String() string {
    return fmt.Sprintf("Book: %s - %s", b.Title, b.Author)
}

// Declare a Count type which satisfies the fmt.Stringer interface.
type Count int

func (c Count) String() string {
```

```

    return strconv.Itoa(int(c))
}

// Declare a WriteLog() function which takes any object that satisfies
// the fmt.Stringer interface as a parameter.
func WriteLog(s fmt.Stringer) {
    log.Print(s.String())
}

func main() {
    // Initialize a Count object and pass it to WriteLog().
    book := Book{"Alice in Wonderland", "Lewis Carroll"}
    WriteLog(book)

    // Initialize a Count object and pass it to WriteLog().
    count := Count(3)
    WriteLog(count)
}

```

This is pretty cool. In the `main` function we've created different `Book` and `Count` types, but passed both of them to the *same* `WriteLog()` function. In turn, that calls their relevant `String()` functions and logs the result.

If you [run the code](#), you should get some output which looks like this:

```

2009/11/10 23:00:00 Book: Alice in Wonderland - Lewis Carroll
2009/11/10 23:00:00 3

```

I don't want to labor the point here too much. But the key thing to take away is that by using an interface type in our `WriteLog()` function declaration, we have made the function agnostic (or flexible) about the exact *type* of object it receives. All that matters is *what methods it has*.

Why are they useful?

There are all sorts of reasons that you might end up using an interface in Go, but in my experience the three most common are:

1. To help reduce duplication or boilerplate code.
2. To make it easier to use mocks instead of real objects in unit tests.
3. As an architectural tool, to help enforce decoupling between parts of your codebase.

Let's step through these three use-cases and explore them in a bit more detail.

Reducing boilerplate code

OK, imagine that we have a `Customer` struct containing some data about a customer. In one part of our codebase we want to write the customer information to a `bytes.Buffer`, and in another part of our codebase we want to write the customer information to an `os.File` on disk. But in both cases, we want to serialize the customer struct to JSON first.

This is a scenario where we can use Go's interfaces to help reduce boilerplate code.

The first thing you need to know is that Go has an `io.Writer` interface type which looks like this:

```
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

And we can leverage the fact that both `bytes.Buffer` and the `os.File` type satisfy this interface, due to them having the `bytes.Buffer.Write()` and `os.File.Write()` methods respectively.

Let's take a look at a simple implementation:

```
package main

import (
    "bytes"
    "encoding/json"
    "io"
    "log"
    "os"
)

// Create a Customer type
type Customer struct {
    Name string
    Age  int
}

// Implement a WriteJSON method that takes an io.Writer as the parameter.
// It marshals the customer struct to JSON, and if the marshal worked
// successfully, then calls the relevant io.Writer's Write() method.
func (c *Customer) WriteJSON(w io.Writer) error {
    js, err := json.Marshal(c)
    if err != nil {
        return err
    }

    _, err = w.Write(js)
    return err
}

func main() {
    // Initialize a customer struct.
    c := &Customer{Name: "Alice", Age: 21}
```

```

// We can then call the WriteJSON method using a buffer...
var buf bytes.Buffer
err := c.WriteJSON(&buf)
if err != nil {
    log.Fatal(err)
}

// Or using a file.
f, err := os.Create("/tmp/customer")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

err = c.WriteJSON(f)
if err != nil {
    log.Fatal(err)
}
}

```

Of course, this is just a toy example (and there are other ways we could structure the code to achieve the same end result). But it nicely illustrates the benefit of using an interface – we can create the `Customer.WriteJSON()` method once, and we can call that method any time that we want to write to something that satisfies the `io.Writer` interface.

But if you're new to Go, this still begs a couple of questions: *How do you know that the `io.Writer` interface even exists? And how do you know in advance that `bytes.Buffer` and `os.File` both satisfy it?*

There's no easy shortcut here I'm afraid – you simply need to build up experience and familiarity with the interfaces and different types in the standard library. Spending time thoroughly reading the standard library documentation, and

looking at other people's code will help here. But as a quick-start I've included a list of some of the most useful interface types at the [end of this post](#).

But even if you don't use the interfaces from the standard library, there's nothing to stop you from creating and using *your own interface types*. We'll cover how to do that next.

Unit testing and mocking

To help illustrate how interfaces can be used to assist in unit testing, let's take a look at a slightly more complex example.

Let's say you run a shop, and you store information about the number of customers and sales in a PostgreSQL database. You want to write some code that calculates the sales rate (i.e. sales per customer) for the past 24 hours, rounded to 2 decimal places.

A minimal implementation of the code for that could look something like this:

```
↳ File: main.go
```

```
package main
```

```
import (  
    "fmt"  
    "log"  
    "time"  
    "database/sql"  
    _ "github.com/lib/pq"  
)
```

```
type ShopDB struct {  
    *sql.DB  
}
```

```

func (sdb *ShopDB) CountCustomers(since time.Time) (int, error) {
    var count int
    err := sdb.QueryRow("SELECT count(*) FROM customers WHERE timestamp > $1",
        return count, err
}

func (sdb *ShopDB) CountSales(since time.Time) (int, error) {
    var count int
    err := sdb.QueryRow("SELECT count(*) FROM sales WHERE timestamp > $1", sin
        return count, err
}

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    shopDB := &ShopDB{db}
    sr, err := calculateSalesRate(shopDB)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf(sr)
}

func calculateSalesRate(sdb *ShopDB) (string, error) {
    since := time.Now().Add(-24 * time.Hour)

    sales, err := sdb.CountSales(since)
    if err != nil {
        return "", err
    }

    customers, err := sdb.CountCustomers(since)
    if err != nil {
        return "", err
    }

```

```
    rate := float64(sales) / float64(customers)
    return fmt.Sprintf("%.2f", rate), nil
}
```

Now, what if we want to create a unit test for the `calculateSalesRate()` function to make sure that the math logic in it is working correctly?

Currently this is a bit of a pain. We would need to set up a test instance of our PostgreSQL database, along with setup and teardown scripts to scaffold the database with dummy data. That's quite lot of work when all we really want to do is test our math logic.

So what can we do? You guessed it – interfaces to the rescue!

A solution here is to create our own interface type which describes the `CountSales()` and `CountCustomers()` methods that the `calculateSalesRate()` function relies on. Then we can update the signature of `calculateSalesRate()` to use this custom interface type as a parameter, instead of the concrete `*ShopDB` type.

Like so:

```
↳ File: main.go
```

```
package main
```

```
import (
    "database/sql"
    "fmt"
    "log"
    "time"
```

```

    _ "github.com/lib/pq"
)

// Create our own custom ShopModel interface. Notice that it is perfectly
// fine for an interface to describe multiple methods, and that it should
// describe input parameter types as well as return value types.
type ShopModel interface {
    CountCustomers(time.Time) (int, error)
    CountSales(time.Time) (int, error)
}

// The ShopDB type satisfies our new custom ShopModel interface, because it
// has the two necessary methods -- CountCustomers() and CountSales().
type ShopDB struct {
    *sql.DB
}

func (sdb *ShopDB) CountCustomers(since time.Time) (int, error) {
    var count int
    err := sdb.QueryRow("SELECT count(*) FROM customers WHERE timestamp > $1",
        since)
    return count, err
}

func (sdb *ShopDB) CountSales(since time.Time) (int, error) {
    var count int
    err := sdb.QueryRow("SELECT count(*) FROM sales WHERE timestamp > $1", since)
    return count, err
}

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    shopDB := &ShopDB{db}
    sr, err := calculateSalesRate(shopDB)
    if err != nil {
        log.Fatal(err)
    }
}

```

```

    }
    fmt.Printf(sr)
}

// Swap this to use the ShopModel interface type as the parameter, instead of
// concrete *ShopDB type.
func calculateSalesRate(sm ShopModel) (string, error) {
    since := time.Now().Add(-24 * time.Hour)

    sales, err := sm.CountSales(since)
    if err != nil {
        return "", err
    }

    customers, err := sm.CountCustomers(since)
    if err != nil {
        return "", err
    }

    rate := float64(sales) / float64(customers)
    return fmt.Sprintf("%.2f", rate), nil
}

```

With that done, it's straightforward for us to create a mock which satisfies our `ShopModel` interface. We can then use that mock during unit tests to test that the math logic in our `calculateSalesRate()` function works correctly. Like so:

↳ File: main_test.go

```

package main

import (
    "testing"
    "time"
)

type MockShopDB struct{}

```

```

func (m *MockShopDB) CountCustomers(_ time.Time) (int, error) {
    return 1000, nil
}

func (m *MockShopDB) CountSales(_ time.Time) (int, error) {
    return 333, nil
}

func TestCalculateSalesRate(t *testing.T) {
    // Initialize the mock.
    m := &MockShopDB{}
    // Pass the mock to the calculateSalesRate() function.
    sr, err := calculateSalesRate(m)
    if err != nil {
        t.Fatal(err)
    }

    // Check that the return value is as expected, based on the mocked
    // inputs.
    exp := "0.33"
    if sr != exp {
        t.Fatalf("got %v; expected %v", sr, exp)
    }
}

```

You could run that test now, everything should work fine.

Application architecture

In the previous examples, we've seen how interfaces can be used to decouple certain parts of your code from relying on concrete types. For instance, the `calculateSalesRate()` function is totally flexible about what you pass to it – the only thing that matters is that it satisfies the `ShopModel` interface.

You can extend this idea to create decoupled 'layers' in larger projects.

Let's say that you are building a web application which interacts with a database. If you create an interface that describes the exact methods for interacting with the database, you can refer to the interface throughout your HTTP handlers instead of a concrete type. Because the HTTP handlers only refer to an interface, this helps to decouple the HTTP layer and database-interaction layer. It makes it easier to work on the layers independently, and to swap out one layer in the future without affecting the other.

I've written about this pattern in [this previous blog post](#), which goes into more detail and provides some practical example code.

What is the empty interface?

If you've been programming with Go for a while, you've probably come across the *empty interface type*: `interface{}`. This can be a bit confusing, but I'll try to explain it here.

At the start of this blog post I said:

- › *An interface type in Go is kind of like a definition. It defines and describes the exact methods that some other type must have.*

The empty interface type essentially *describes no methods*. It has no rules. And because of that, it follows that any and every object satisfies the empty interface.

Or to put it in a more plain-English way, the empty interface type `interface{}` is kind of like a wildcard. Wherever you see it in a declaration (such as a variable, function parameter or struct field) you can use an object *of any type*.

Take a look at the following code:

```
package main

import "fmt"

func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"
    person["age"] = 21
    person["height"] = 167.64

    fmt.Printf("%+v", person)
}
```

In this code snippet we initialize a `person` map, which uses the `string` type for keys and the empty interface type `interface{}` for values. We've assigned three different types as the map values (a `string`, `int` and `float32`) – and that's OK. Because objects of any and every type satisfy the empty interface, the code will work just fine.

You can [give it a try here](#), and when you run it you should see some output which looks like this:

```
map[age:21 height:167.64 name:Alice]
```

But there's an important thing to point out when it comes to retrieving and using a value from this map.

For example, let's say that we want to get the `"age"` value and increment it by 1. If you write something like the following code, it will fail to compile:


```

package main

import "log"

func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"
    person["age"] = 21
    person["height"] = 167.64

    person["age"] = person["age"] + 1

    fmt.Printf("%+v", person)
}

```

And you'll get the following error message:

```
invalid operation: person["age"] + 1 (mismatched types interface {} and int)
```

This happens because the value stored in the map takes on the type `interface{}`, and ceases to have it's original, underlying, type of `int`. Because it's no longer an `int` type we cannot add 1 to it.

To get around this this, you need to type assert the value back to an `int` before using it. Like so:

```

package main

import "log"

func main() {
    person := make(map[string]interface{}, 0)

    person["name"] = "Alice"

```

```

    person["age"] = 21
    person["height"] = 167.64

    age, ok := person["age"].(int)
    if !ok {
        log.Fatal("could not assert value to int")
        return
    }

    person["age"] = age + 1

    log.Printf("%+v", person)
}

```

If you [run this now](#), everything should work as expected:

```
2009/11/10 23:00:00 map[age:22 height:167.64 name:Alice]
```

So when should you use the empty interface type in your own code?

The answer is *probably not that often*. If you find yourself reaching for it, pause and consider whether using `interface{}` is really the right option. As a general rule it's clearer, safer and more performant to use concrete types – or non-empty interface types – instead. In the code snippet above, it would have been more appropriate to define a `Person` struct with relevant typed fields similar to this:

```

type Person struct {
    Name    string
    Age     int
    Height  float32
}

```


But that said, the empty interface is useful in situations where you need to accept and work with unpredictable or user-defined types. You'll see it used in a number of places throughout the standard library for that exact reason, such as in the [gob.Encode](#), [fmt.Print](#) and [template.Execute](#) functions.

Common and useful types

Lastly, here's a short list of some of the most common and useful interfaces in the standard library. If you're not familiar with them already, then I recommend taking out a bit of time to look at the relevant documentation for them.

- [builtin.Error](#)
- [fmt.Stringer](#)
- [io.Reader](#)
- [io.Writer](#)
- [io.ReadWriteCloser](#)
- [http.ResponseWriter](#)
- [http.Handler](#)

There is also a longer and more comprehensive listing of standard libraries available in [this gist](#).

 *If you enjoyed this article, you might like to check out my [recommended tutorials list](#) or check out my books [Let's Go](#) and [Let's Go Further](#), which teach you everything you need to know about how to build professional production-ready web applications and APIs with Go.*

Filed under: [#golang](#) [#tutorial](#)

© Copyright 2013-2023 Alex Edwards

Code snippets are [MIT licensed](#)

[Disclaimer](#)

Enjoy the rest of your Monday!