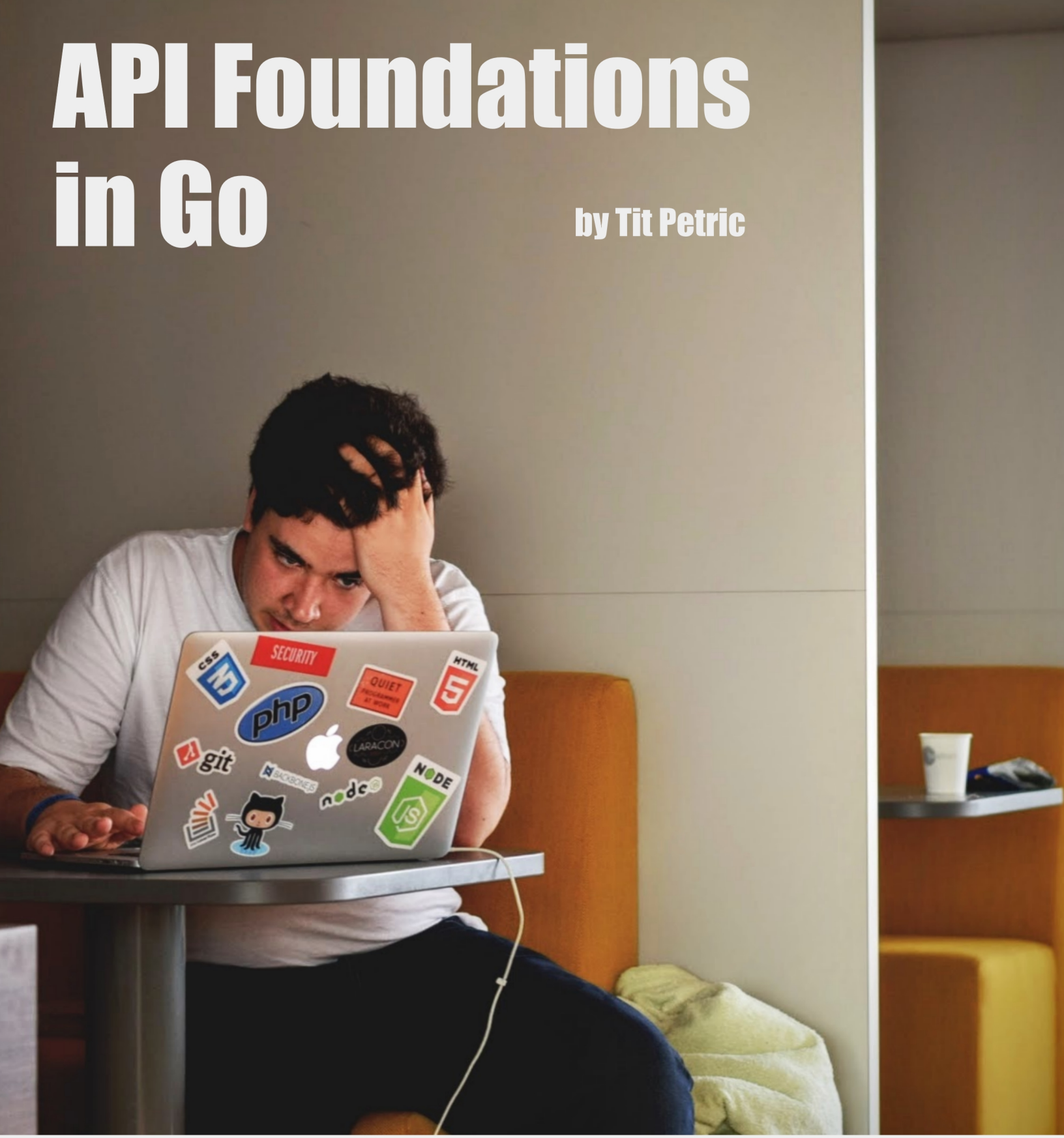
# API Foundations in Go

by Tit Petric



# API foundations in Go

## You've mastered PHP and are looking at Node.js? Skip it and try Go.

Tit Petric

This book is for sale at http://leanpub.com/api-foundations

This version was published on 2019-01-15


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Tweet This Book!

Please help Tit Petric by spreading the word about this book on Twitter!

The suggested hashtag for this book is #apifoundations.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#apifoundations

*I'm dedicating this book to my wife, Anastasia. Even if you say that you don't suffer, I'm sure that I'm killing your inner child by explaining what Docker is. I'm sorry for that.*

# Contents

# Introduction

## About me

I'm one of those people with about two decades of programming experience under my belt. I started optimizing code in the '90s, discovered PHP in the 2000s, and have built several large-scale projects to date, all while discovering other programming language families like Java, Node.js and ultimately, Go.

I have built numerous APIs for my own content management products. Several products I've been involved with as a lead developer have been sold and are used in multiple countries. I've written a professional, dedicated API framework, which doubles as a software development kit for the Slovenian national TV and Radio station website, RTV Slovenia. I've also been a speaker at several local PHP user group events and conferences.

## Why Go?

Go has been on my list for a while now. About a year or two ago, my coworker created a protocol converter that emulates a Memcache server, using Redis as the backend datastore. Since our biggest project has a code base built up over some 13 years, it was preferable to keep the working code as-is, and just replace the software around it as needed. Go made this possible.

One of the reasons for choosing Go was the constant comparison of Go with Node. Node has a much more vocal community, which seems to religiously advocate it. We have carried out several tests in recent months, and Node, while not very hard to start development with, had poorer performance than pretty much everything except PHP. I'm not saying that Go is better than Node, or that anything is better than anything else, but from what we've seen, it seems advisable to skip Node.js and go straight to Go. This might change as ES6 and ES7 get more traction - but there are immediate benefits of switching to Go. If you don't want to move from Node.js, this book is not for you. If you have an open mind - find out what Go can do.

## Who is this book for?

This book is for senior developers who might not have had a chance to try Go, but are familiar with concepts of API development in languages like PHP or Node. Any reader must have a good understanding of REST APIs and server architecture. While I'll try to make everything as clear as possible, realize that if you're a novice programmer, there might be a gap between what you know, and what I'm trying to explain here.

I'm not going to be explaining the Go programming language in detail. I'm going to be diving head first into using it with just a note here and there. This is why familiarity and strong knowledge of programming concepts are required.

In the book, I will cover these subjects:

1. Setting up your environment
2. Data structures
3. Organizing your code
4. Encoding and decoding JSON
5. Serving HTTP requests
6. Parallel fetching of data
7. Using external services (Redis)
8. Using external services (MySQL)
9. Test driven API development
10. Your first API
11. Running your API in production
12. Resources

Covering these concepts should give you a strong foundation for your API implementation. The book doesn't try to teach you Go; the book tries to give you a strong software foundation for APIs, using Go.

## How should I study it?

Through the book, I will present several examples on how to do common things when developing APIs. The examples are published on GitHub, you can find the link in the last chapter of the book.

You should follow the examples in the book, or you can look at each chapter individually, just to cover the knowledge of that chapter. The examples are stand-alone, but generally build on work from previous chapters.

---

# Setting up your environment

Setting up a development environment, as well as a production environment, is an important topic today. While spinning up a virtual machine and installing software by hand is perhaps the accepted way of doing things, recently I've learned to systematize my development by using Docker containers.

The biggest benefit of Docker containers is a "zero install" way of running software - as you're about to run a container, it downloads the image containing all the software dependencies you need. You can take this image and copy it to your production server, where you can run it without any change in environment.

## Networking

When you have docker set up, we will need to create a network so the services that we're going to use can talk to each other. Creating a network is simple, all you need to do is run the following command:

```
$ docker network create -d bridge --subnet 172.25.0.0/24 party
```

This command will create a network named **party** on the specified subnet. All docker containers which will run on this network will have connectivity to each other. That means that when we run our Go container, it will be able to connect to another Redis container on the same network.

## Setting up a runner for your Go programs

There is an official Go image available on Docker. Getting a Docker image and running it is very simple, requiring just one line:

```
$ docker run --net=party -p 8080:80 --rm=true -it -v `pwd`:/go/src/app -w /go/src/ap\
p golang go "$@"
```

Save this code snippet as the file 'go', make it executable and copy it to your execution path (usually /usr/local/bin is reserved for things like this).

Let's quickly go over the arguments provided:

- **–net=party** - runs the container on the shared network

- **-p** - network forwarding from host:8080 to container:80 (HTTP server)
- **–rm=true** - when the container stops, clean up after it (saves disk space)
- **-v option** - creates a volume from the current path (pwd) in the container
- **”$@”** - passes all arguments to go to the container application

Very simply, what this command does is run a Docker container in your current folder, execute the `go` binary in the container, and clean up after itself when the program finishes.

> *Note*: as we only expose the current working path to the container, it limits access to the host machine - if you have code outside the current path, for example in “..” or “/usr/share”, this code will not be available to the container.

An example of running go would be:

```
$ go version
go version go1.6 linux/amd64
```

And, what we will do through most of this book is run the following command:

- **go run [file]** - compile and run Go program

```
$ go run hello_world.go
Hello world!
```

**A minimal example of a Go program**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    fmt.Printf("Hello world!\n")
7  }
```

# Setting up Redis

We will also use Docker to run Redis, which is as simple as setting up Go. Redis is an in-memory data structure store. It provides functionality to store and retrieve data in various structures, beyond a simple key-value database like Memcache. We will use this service later in the book when implementing our example API endpoints.

To run an instance of Redis named ‘redis’:

```
$ docker run --restart=always -h redis --name redis --net=party -d redis
```

Just like Go, Redis provides an official build on the Docker Hub. Interacting with Redis will be covered in detail in later chapters.

## Other services

In addition to Go and Redis, other services are also available on the Docker Hub[1]. Depending on your use case, you might want to install additional services via docker.

Popular projects which I use from Docker on a daily basis:

- nginx (and nginx-extras flavours)
- Percona (MySQL, MariaDB) - also covered later in the book
- letsencrypt
- redis
- samba
- php

Docker is a very powerful tool which gives you all the software you might need. It's very useful also for testing, as you can use it to set up a database, populate it with test data, and then tear down and clean up after it. It's a very convenient way to run programs that are isolated from your actual environment, and may only be active temporary.

[1] https://hub.docker.com/

# Data structures

Defining and handling data structures is a key activity in any programming language. When talking about object oriented programming, it's worth noting some differences between Go and other programming languages.

- Go doesn't have classes, it has structs
- Methods are bound to a struct, not declared within one
- The "interface" type can be an intersection of many or all types
- Packages behave like namespaces, but everything in a package is available

In short, it means that the functions you'll define for your structs will be declared outside of the struct they are working on. If you want to declare several types, the only way to assign them to a same variable, without many complications, is to declare a common interface.

## Declaring structs

When we define a structure in Go, we set types for every member. Let's say we would like to define the usual "Petstore", which in turn has a list of pets. You could define the structure like this:

```
type Petstore struct {
  Name string
  Location string
  Dogs []*Pet
  Cats []*Pet
}

type Pet struct {
  Name string
  Breed string
}
```

The example is simple, in the sense that I'm not defining a "Pet", which can be a "Dog", or can be a "Cat". I'm just using one type for all.

> *Note*: Members of defined structures begin with an upper case letter. This means that they are visible outside the package they are defined in. All members, which you will encode to JSON in the next chapter, need to be public, that is - start with an upper case letter. The same applies to functions that you expose in packages.

# Casting structs

We could declare a Dog and Cat struct, and could cast one to one from the other. A requirement for this is that the Dog and Cat types *must have identical underlying types*. So, if I was to extend the above example, I would copy the Pet struct into Dog and Cat so they are identical.

If they are not identical, the value can't be cast.

**Error when casting**

```
type Dog struct {
  name string
}
type Cat struct {
  name           string
  hypoallergenic bool
}

func main() {
  dog := Dog{name: "Rex"}
  cat := Cat(dog)
}
```

The above example results in:

```
./type1.go:12: cannot convert dog (type Dog) to type Cat
```

Even if "Cat" has all the properties of "Dog", type conversion is not possible. It makes no sense to declare strong types of individual animals, if casting to a generic type is not possible. But we can assign these kind of strong types to a common interface type.

# Declaring interfaces

An interface defines zero or more methods that need to be implemented on types/objects that can be assigned to that interface. If we extend the above example to declare explicit types for Cat and Dog, we will end up with something like this:

**Structure for an agnostic pet list**

```
type Dog struct {
  name  string
  breed string
}
type Cat struct {
  name           string
  hypoallergenic bool
}
type Petstore struct {
  name string
  pets []interface{}
}
```

We are using the declaration of `pets` as "interface{}". The interface doesn't define any common methods, so anything can be assigned to it. Usually, one would implement getter or setter methods which should be common to all the types an interface can hold. With our example, we could have declared a function `GetName` that would return the name of the Pet:

**Structure for an agnostic pet list**

```
type Pet interface {
  getName() string
}

func (d Dog) getName() string {
  return d.name
}
func (c Cat) getName() string {
  return c.name
}
```

# Abusing interfaces

Interfaces are a powerful and flexible way to work with objects of varying types. You should however tend to prefer static type declarations when possible - it makes the code faster and more readable. The Go compiler catches a lot of your common errors with static analysis. If you're using interfaces, you're exposing yourself to risks and errors which the compiler cannot catch.

All the variables which have been declared in the object are unreachable from the interface and will result in an error. But that doesn't mean that you can't access them.

If you absolutely must, you can use a feature called "reflection". Reflection exposes the members of any interface via an API.

**"Hello world!" example using Reflection**

```
package main

import "fmt"
import "reflect"

type Test1 struct {
  A string
}

func main() {
  var t interface{}
  t = Test1{"Hello world!"}

  data := reflect.ValueOf(t)
  fmt.Printf("%s\n", data.FieldByName("A"))
}
```

Interfaces can be a powerful part of Go, but care should be taken not to overuse them. They make many tasks simpler and can be a powerful tool when they are used correctly.

> Reflection is used by many packages to provide generic function interfaces, like the encoding/json package does, which we will use in a later chapter.

# Embedding and composition

You can embed types into a struct by listing them in the struct declaration. This makes all the values and functions of the embedded struct available for use on your struct. For example, the `sync.Mutex` struct implements `Lock` and `Unlock` functions.

```
type Services struct {
  sync.Mutex
  ServiceList map[string]Service
}

func (r *Services) Add(name string, service Service) {
  r.Lock()
  r.ServiceList[name] = service
  r.Unlock()
}
```

This pattern enables composition. You can build your APIs for a larger program from smaller structures that can deal with specific problem domains. The example also shows a beneficial side-effect: you can reason that the embedded `sync.Mutex` is used to lock your structure fields below the embed.

You can use embedding to your advantage with, for example, `sync.Mutex`, `sync.RWMutex` or `sync.WaitGroup`. You can actually embed *many* structs, so your structure may perform the functions of them all.

An example from a project I'm working on uses two embedded structs:

```
type RunQueue struct {
  sync.RWMutex
  sync.WaitGroup
  // ...
  flagIsDone  bool
}
func (r *RunQueue) Close() {
  r.Lock()
  defer r.Unlock()
  r.flagIsDone = true
}
func (r *RunQueue) IsDone() bool {
  r.RLock()
  defer r.RUnlock()
  return r.flagIsDone
}
```

## Leveraging `sync.RWMutex`

The declaration of the RunQueue struct above leverages the `sync.RWMutex` to provide synchronous access to the object from many goroutines. A goroutine may use `Close` to finish the execution of the goroutine queue. Each worker in the queue would call `IsDone` to check if the queue is still active.

### Leveraging `sync.WaitGroup`

The RunQueue struct leverages a `sync.WaitGroup` to provide queue clean up and statistics, such as elapsed time. While I can't provide all the code, the basic usage is as follows:

```
func (r *RunQueue) Runner() {
  fmt.Printf("Starting %d runners\n", runtime.NumCPU())
  for idx := 1; idx <= runtime.NumCPU(); idx++ {
    go r.runOnce(idx)
  }
}
func NewRunQueue(jobs []Command) RunQueue {
  q := RunQueue{}
  for idx, job := range jobs {
    if job.SelfId == 0 {
      q.Dispatch(&jobs[idx])
    }
  }
  q.Add(len(q.jobs)) // sync.WaitGroup
  return q
}
runnerQueue := NewRunQueue(commands)
go runnerQueue.Finisher()
go runnerQueue.Runner()
runnerQueue.Wait() // sync.WaitGroup
```

The main idea of the program I'm building is that it starts `runtime.NumCPU()` runners, which handle execution of a fixed number of commands. The `WaitGroup` comes into play very simply:

```
NewRunQueue calls *wg.Add(number of jobs)
Individual jobs are processed with RunQueue.runOnce, and they call *wg.Done()
RunnerQueue.Wait() (*wg.Wait()) will wait until all jobs have been processed
```

## Limiting goroutine parallelization

At one point I struggled to create a queue manager, which would parallelize workloads to a fixed limit of parallel goroutines. My idea was to register a slot manager, which would provide a pool of running tasks. If no pool slot is available, it'd sleep for a few seconds before trying to get a slot again. It was frustrating.

Just look at the loop from the `Runner` function above:

```
for idx := 1; idx <= runtime.NumCPU(); idx++ {
  go r.runOnce(idx)
}
```

This is an elegant way to limit parallelization to N routines. There is no need to bother yourself with some kind of routine allocation pool structs. The `runOnce` function should only do a few things:

1. Listen for new jobs in an infinite loop, reading jobs from a channel
2. Perform the job without new goroutines

The reason to read the jobs from a channel is that the read from a channel is a blocking operation. The function will just wait there until a new job appears on the channel it's reading from.

```
func (r *RunQueue) runOnce(idx int) {
  for {
    queueJob, ok := <-r.runQueue
    if !ok {
      return
    }
    // run tasks
[...]
```

The job needs to be executed without a goroutine, or with nested `*WaitGroup.Wait()` call. The reason for this should be obvious - as soon as you start a new goroutine, it gets executed in parallel and the `runOnce` function reads the next job from the queue. This means that the limit on the number of tasks running in parallel would not be enforced.

# Slices

A Slice is a Go-specific data type, which consists of:

1. a pointer to an array of values,
2. the capacity of the array,
3. the length of the array

If you're working with slices, which you mostly are, you'll inevitably find yourself in a situation where you'll have to merge two slices into one. The naïve way of doing this is something like the following:

```
alice := []string{"foo", "bar"}
bob := []string{"verdana", "tahoma", "arial"}
for _, val := range bob {
    alice = append(alice, val)
}
```

A slightly more idiomatic way of doing this is:

```
alice = append(alice, bob...)
```

The `ellipsis` or variadic argument expands `bob` in this case to all its members, effectively achieving the same result, without that loop. You can read more about it in the official documentation: Appending to and copying slices[2]

You could use it to wrap logic around `log.Printf` that, for example, trims and appends a newline at the end of the format string.

```
import "log"
import "strings"

func Log(format string, v ...interface{}) {
    format = strings.TrimSuffix(format, "\n") + "\n"
    log.Printf(format, v...)
}
```

Since slices are basically a special type of pointer, this means there are a number of scenarios where you may inadvertently modify a slice without intending to. Thus, it's important to know how to manipulate slices.

## The slice operator

The first thing one should be aware of is that the slice operator, *does not copy the data to a newly-created slice*. Not being aware of this fact can lead to unexpected results:

```
a := []string{"r","u","n"}
b := a[1:2]
b[0] = "a"
fmt.Println(a)
```

[2] https://golang.org/ref/spec#Appending_and_copying_slices

Does it print `[r u n]` or `[r a n]`? Since the slice `b` is not a copy of the slice, but just a slice with a modified pointer to the array, the above will print `[r a n]`.

As explained, this is because the slice operator just provides a new slice with an updated reference to the same array as used by the original slice. From the official reference:

> Slicing does not copy the slice's data. It creates a new slice value that points to the original array. This makes slice operations as efficient as manipulating array indices. Therefore, modifying the elements (not the slice itself) of a re-slice modifies the elements of the original slice.

Source: Go Blog - Slices usage and internals[3].

# Allocation by append

Appending to a slice is simple enough. As mentioned, the slice has a length which you can get with a call to `len()`, and a capacity which you can get with a call to `cap()`.

```
a := []string{"r", "u", "n"}
fmt.Println(a, len(a), cap(a))
a = append(a, []string{"e"}...)
fmt.Println(a, len(a), cap(a))
a = a[1:len(a)]
fmt.Println(a, len(a), cap(a))
```

The expected output would be:

```
[r u n] 3 3
[r u n e] 4 4
[u n e] 3 3
```

Of course, that's not how `append` works. Append will double the existing capacity. This means you'll end up with output like this:

```
[r u n] 3 3
[r u n e] 4 6
[u n e] 3 5
```

If you wanted the previous result, you would have to create your own function, which would allocate only the required amount of items into a new slice, and then copy over the data from the source slice. This function might look something like this:

[3] https://blog.golang.org/go-slices-usage-and-internals

```
func suffix(source []string, vars ...string) []string {
  length := len(source) + len(vars)
  ret := make([]string, length, length)
  copy(ret, source)
  index := len(source)
  for k, v := range vars {
    ret[index+k] = v
  }
  return ret
}

func main() {
  a := []string{"r", "u", "n"}
  fmt.Println(a, len(a), cap(a))
  a = suffix(a, []string{"e"}...)
  fmt.Println(a, len(a), cap(a))
  a = a[1:len(a)]
  fmt.Println(a, len(a), cap(a))
}
```

## Copying slices

As explained above, “slicing does not copy the slice’s data”. This may sometimes have unintended consequences. This applies not only to slicing, but also to passing slices into functions.

```
func flip(source []string) {
  source[1] = "a"
}

func main() {
  a := []string{"r", "u", "n"}
  flip(a)
  fmt.Println(a)
}
```

The above example will print `[r a n]`. This is unfortunate, because people intuitively expect slices to behave much like structs do. Passing a struct will create a copy. A slice will still point at the same memory that holds the data. Even if you pass a slice within a struct, you should be aware of this:

```
type X struct {
  c string
  source []string
}

func flip(x X) {
  x.c = "b"
  x.source[1] = "a"
}

func main() {
  a := X{"a", []string{"r", "u", "n"}}
  flip(a)
  fmt.Println(a)
}
```

The above will print out a `{a [r a n]}`. Slices always behave as if they are passed by reference. In a way they are, as one of the parts of the slice is a pointer to the array of data it holds.

## Using slices in channels

If you're using buffered channels, and are trying to be smart with slices, reusing a slice to keep allocations down, then you might find that this buffered channel is basically filled with the same slice over and over. As the slice contains the pointer to the array holding the data, the following example will have unexpected results:

```
func main() {
  c := make(chan []string, 5)

  go func() {
    item := []string{"hello"}
    for i := 0; i < 5; i++ {
      item[0] = fmt.Sprintf("hello %d", i)
      c <- item
      //time.Sleep(100 * time.Millisecond)
    }
  }()

  for i := 0; i < 5; i++ {
    item := <-c
    fmt.Println(item)
```

```
16    }
17  }
```

The output is:

```
[hello 4]
[hello 4]
[hello 4]
[hello 4]
[hello 4]
```

If you uncomment the `time.Sleep` command in the goroutine, you will most likely get the correct result:

```
[hello 0]
[hello 1]
[hello 2]
[hello 3]
[hello 4]
```

At any time when the consumer is reading from the channel, the retrieved slice will be identical, because only the item in the underlying array is changing. The only solution for this, I believe, is either finding a way to work with an unbuffered channel (a consumer must *always* be waiting to read from the channel), or to explicitly copy the slice which is being put into the channel, like in this playground example[4].

[4] https://play.golang.org/p/ORPyzf0zwY