Why Go?

Interpreted, dynamically typed languages such as Python, Ruby or Javascript are: easy to use, allow fast development and have a great number of libraries. Unfortunately, they are not very good when it comes to performance.

For this reason, I felt the need to look for something different for some projects. However, I didn't want to lose the simplicity other languages gave me. Essentially, I wanted something that could give me:

* Good performance
* Easy concurrency
* Easy error handling
* Safety
* Fast development time
* Good libraries to ease my development

Go was the obvious choice - I had previously worked with it and knew I could achieve all of these points.

So what is Go?

From the [Go documentation page](https://golang.org/doc/faq#creating_a_new_language)

Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language, with the efficiency and safety of a statically typed,compiled language.

An attempt which I believe was mostly successful. But let's dive into some points to understand Go's beauty.

Simple and easy

Go is a "simple" language - it only contains 25 keywords (Python 3.5.0: 33, Ruby 2.2.0: 41, ANSI C: 32). Obviously, a small number of keywords doesn't necessarily mean it's easy - [Brainfuck](https://en.wikipedia.org/wiki/Brainfuck) only has 8 and its name suggests how easy it is to program in it.

And Go does have trade-offs for using such a small number of keywords. For example, it only has one loop (keyword for), which can be seen has a nice thing since you really only have to know one word. But it can take several forms:

* for condition behaves like a while
* for i,v := range list ranges over a list
* for i=0;i<10;i++ for a C style classical for

which might confuse/annoy some people.

However, I do believe Go is easy to understand and learn, and you can see it for yourself by doing the   
[Golang Tour](https://tour.golang.org/welcome/1), which will teach you the essentials.

Standard Library

Furthermore, Go has got a robust standard library which allows quick and easy development of common essential tasks:

* net (http, mail)
* archives (zip, tar, gzip)
* encodings (json, csv)
* cryptography (aes, des, md5)
* html template system
* generic sql drivers
* logging system
* and many others...

Easy to read

Any programmer can tell you that reading other people's (including your past self) code is a daunting, hair-ripping task. However, it's usually inevitable, so it is nice to see a language which in a way facilitates this.

What I'm talking about is the gofmt tool which formats all Go code the same way, thus making it easier to: read,   
write, maintain, and eliminates endless discussions and Stackoverflow questions about whether one should use spaces or tabs, brace positioning, etc...

Sure it might not be the kind of formatting you are accustomed to, or even like, but, at least, it's always the same. Whether I am reading library A, B, C, or D the format is always the same which facilitates source code reading.

Go tools

gofmt is not the only tool which Go has to offer, many more are available. To name some:

* golint to check for style violations
* go vet to check for programming errors
* go test to test the code
* go tool cover to get coverage reports (given tests)
* go doc to generate documentation

This great tooling ecosystem makes it so that I don't have to worry about many things when patching a bug for some library. Is my formatting correct? Is the testing library I am using the correct one? Is the style correct?   
Being able to just focus on the logic instead of all those details when contributing to a project is one of the aspects I love about Go. I hope to see more languages adopt this in the future.

Let's get to the code

But enough talk, let's see an example of how easy and faster Go can be when compared to other options

REST API

Recently I needed to develop a REST API and decided to choose Go because I wanted good performance.   
At the time I went with the [Gin Framework](https://gin-gonic.github.io/gin/) which would result in the following code:

package main

import "github.com/gin-gonic/gin"

func main() {

r := gin.Default()

r.GET("/", func(c \*gin.Context) {

c.String(200, "Hello World!")

})

r.Run(":8080")

}

Which as you can see is very straight forward. You setup the framework with default options, setup the route and run it. However, more recently I have started to use the standard http library:

package main

import (

"fmt"

"net/http"

)

func main() {

http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "Hello World!")

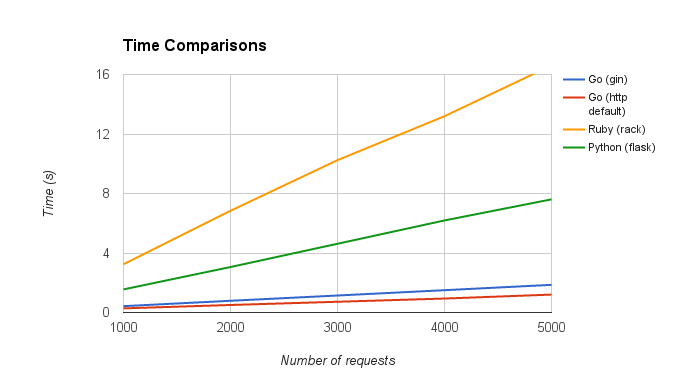
})

http.ListenAndServe(":8080", nil)

}

which is as simple as the framework above.

Furthermore, I was also interested in checking out the performance gains against Python and Ruby (languages I would probably use by default for this kind of task).

Figure 1: Comparison of performance for single page http servers

As you can see in the picture above Go was able to perform much better than the python and ruby examples. Small note: this was just a quick comparison, it is by no means a proper analysis, as a sub-note always   
be wary of benchmarks you find on the web.

Obviously, I wanted to do more complex things, but this small example shows how powerful Go can be whilst keeping things simple for the developer.