UNIVERSITY OF MACEDONIA
GRADUATE PROGRAM
DEPARTMENT OF APPLIED INFORMATICS

DISTRIBUTED COMPUTING IN GO: COMPARATIVE ANALYSIS OF WEB APPLICATION FRAMEWORKS

Master Thesis

of

Evgenios Sochopoulos

Thessaloniki, June 2020

DISTRIBUTED COMPUTING IN GO: COMPARATIVE ANALYSIS OF WEB APPLICATION FRAMEWORKS

Evgenios Sochopoulos

Diploma in Rural and Surveying Engineering, Aristotle University of Thessaloniki, 2017

Master Thesis

submitted for the partial fulfillment of the

DEGREE OF MASTER OF SCIENCE IN APPLIED INFORMATICS

Supervisor
Professor Konstantinos G. Margaritis

Abstract

While the Go programming language it is a relatively new language, it has gained immense popularity in the last years and is now used by many major companies and organizations, especially in the cloud computing field. Go is a great language for developing high-performance, scalable web applications and, as it is an open-source language, many web frameworks have emerged to help the development of such applications.

The main objective of this thesis is to evaluate and compare the features of five selected Go web frameworks, based on certain criteria. It consists of four chapters. The first chapter is an introduction to Go with general information about the origins, the design and its usage, followed by a demonstration of the language's most basic aspects that are accompanied by code examples. The second chapter presents a brief overview of distributed computing, from introductory concepts and communication techniques to the ways software components of distributed systems are organized. In the third chapter, five Go web frameworks are selected along with specific evaluation criteria, based on which the selected frameworks are compared to each other. Finally, the fourth chapter gives a detailed description of the procedure followed in the making of a demo application using each one of the five selected frameworks, which was used for the determination of the criteria for the frameworks comparison.

Keywords: distributed computing, Go, web framework, web application, web application framework, comparative analysis, frameworks comparison, REST

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CDN	Content Delivery Network
CLI	Command Line Interface
CRUD	Create Retrieve Update Delete
CSS	Cascading Style Sheets
CSV	Comma Separated Values
DevOps	Development and Operations
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
JS	JavaScript

JSON	JavaScript Object Notation
JSONP	JavaScript Object Notation with Padding
LIFO	Last In First Out
LLVM	Low Level Virtual Machine
MVC	Model View Controller
ORM	Object Relational Mapping
OS	Operating System
REST	Representational State Transfer
RPC	Remote Procedure Call
SSL	Secure Sockets Layer
SMTP	Simple Mail Transfer Protocol
TLS	Transport Layer Security
TXT	Text
TOML	Tom’s Obvious, Minimal Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language
MCR	Model Controller Routes

1 An Introduction to Go

1.1 The Go Programming Language

1.1.1 Origins

On September 21, 2007, Robert Griesemer, Rob Pike and Ken Thompson started conceiving the first ideas for a new language. A few days were enough to transform these ideas into a plan to do something and also to have a fair idea of what the language would be eventually. The language design continued part-time in parallel with unrelated work. Until January 2008, Thompson had started work on a prototype compiler to explore some ideas of his, which generated C code as its output. By mid-year the language had become a full-time project and had matured enough for a first attempt to create a production compiler. In May 2008, Ian Taylor started working independently on a GCC front-end for Go using the draft specification. Another addition to the team was Russ Cox, who joined in late 2008 and helped move the language and libraries from prototype to reality. Go became an open source project on November 10, 2009. Since then, countless people from the Go community have contributed to the project with their precious ideas, discussions and code [1].

Go was born out of frustration with existing languages and environments for the work that was carried out at Google. Programming had become difficult and one of the reasons was the languages that were being used. At the time of Go’s inception, production software was usually written in C++ or Java, and there was a big frustration by the unneeded complexity required to use the above languages for server programming. Furthermore, computers had become a lot quicker since the first appearance of languages like C, C++ or Java. It was clear that multiprocessors were becoming universal, but no language offered enough help to program easily these processors and also the programming art was not too advanced [1].

So, the choices were either efficient compilation, efficient execution, or ease of programming. All three of those characteristics were not available in the same language. Programmers were choosing ease over safety and efficiency by using dynamically typed languages such as Python and JavaScript rather than C++ or even Java [1].

Google was not the only company troubled by these concerns. After many years of a quiet landscape for programming languages, Go was among the first of several new languages like Rust, Elixir or Swift that have made programming language development an active field again [1].

As a result, Google took a step back and thought how a new programming language would face the most critical issues that were going to dominate software engineering in the years ahead. Go addressed these issues by endeavoring to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. On top of that, it aimed to be modern, suitable for networked and multicore computing with first-class support for concurrency and parallelism, but also to make resource management tractable in a large concurrent program and ensure some sort of safe automatic memory management (garbage collection). But most importantly, working with Go was intended to be fast, meaning it should take at most a few seconds to build a large executable file on a single computer. To meet these goals, the language would require to have an expressive but lightweight type system, concurrency and garbage collection, rigid dependency specification and so on. These features can only be addressed well by a new language and not by libraries or tools [1].

These considerations led to a series of discussions from which Go arose, first as a set of prerequisites and ideas, then as a language. Go also had a more general purpose: to do more to help the working programmer by enabling tooling, automating tedious tasks such as code formatting and making it easier to work on large code bases [1].

1.1.2 Design

The basic syntax of Go is similar to that of the C family, with significant input from the Pascal, Modula and Oberon languages family, regarding declarations and packages. Regarding the concurrency practices, Go was influenced by languages such as Newsqueak and Limbo [1].

Go’s design attempts to reduce the amount of typing in both senses of the word, as well as to reduce clutter and complexity. As a result, there are no forward declarations and no header files and everything is declared exactly once. Syntax is clean, with a small number of keywords. Initialization is truly expressive, easy to use and automatic by using a single declare-and-initialize construct. A radical feature of the language is that there is no type hierarchy. Types just are, without having to announce their relationships. All these simplifications allow the language to be expressive yet comprehensible without sacrificing sophistication [1].

An important principle is to maintain the orthogonality of the concepts. For example, in Go, methods can be implemented for any type and structures represent data while interfaces represent abstraction. Orthogonality makes it easier to understand what happens when many things are combined together [1].

Is Go an object-oriented language? Yes and no at the same time. Although Go has types and methods and allows to program in an object-oriented way, there is no type hierarchy. The "interface" in Go provides a different approach that is really easy to use and more general in some ways. There is also the capability of embedding types in other types to provide something similar-but not identical-to subclassing. In addition, methods in Go are more general than in languages such as C++ or Java, meaning that they can be defined for any kind of data, even built-in types such as integers. It is clear that they are not restricted to structs (the analogous to classes in this case). Another thing is that, the lack of a type hierarchy makes "objects" in Go feel more lightweight than in traditional object-oriented languages languages [1].

Go has an extensive library, called the runtime, which is part of every Go program. The runtime library implements critical features of the Go language: garbage collection, concurrency, stack management and other ones too. Go's runtime could be compared to libc, the C library. However, it is important to understand that Go's runtime does not involve a virtual machine, like the Java runtime for example. Programs written in Go are compiled ahead of time to native machine code (for some variant implementations could also be compiled to JavaScript or WebAssembly). To sum up, although the term "runtime" is most often used to describe the virtual environment in which a program runs, in Go is just the name given to the library that enables the use of critical language services [1].

Apart from having a lot of useful features, Go also lacks some pretty common tools found in many languages: generics, exceptions and assertions. Generics are convenient, but they come at a cost in complexity in the type system and run-time, plus the current Go traits enable a similar behavior to that of generics. Also Go does not have exceptions, because they often result in convoluted code and in the labelling of many ordinary errors as exceptional. Instead, Go offers the multi-value returns, errors as a canonical type and the “defer”, “panic” and “recover” functions for truly exceptional conditions. Last, assertions are absent due to the need of servers and programs in general to be live, even if an error occurs. In addition, they tend to make programmers avoid thinking properly about error handling and reporting [1].

Go was also designed to provide concurrency as a basic component. But concurrency and multi-threaded programming have developed a reputation for difficulty over time. Google’s belief is that, this happens due partly to complex designs like pthreads and partly to overemphasizing on low-level details such as mutexes, condition variables and memory barriers. Higher-level concepts enable a lot simpler code, even if there are still lower level designs, such as mutexes, under the covers [1]. Tony Hoare's Communicating Sequential Processes, or just CSP, is one of the most successful models for providing high-level linguistic support for concurrency. Two well-known languages that stem from CSP are Occam and Erlang. Go's concurrency primitive structures come from a different part of the family tree whose main contribution is the notion of channels as first-class objects, as experience with earlier languages has shown that the CSP model fits very well into a procedural language framework [1].

Goroutines are also part of making concurrency easy to use with the Go language. The main idea is to multiplex independently executing functions-coroutines onto a set of threads. When a coroutine blocks, such as by calling a blocking system call, the run-time automatically moves other coroutines on the same operating system thread to a different, runnable thread so they won't be blocked. The programmers can’t see any of these operations; and this is the point. The result, which Go calls goroutines, can be very cheap, so that they have just a few kilobytes of overhead beyond the memory for the stack. For the stacks to be small, the Go run-time uses resizable, bounded stacks. A newly created goroutine is given a few kilobytes, which is enough most of the time. When it isn't, the run-time grows (or shrinks) the memory for storing the stack automatically, allowing many goroutines to live in a modest amount of memory. So it could be practical to create a large number of goroutines in the same address space. If goroutines were just the classic threads, system resources would run out at a much smaller number of existing goroutines [1].

1.1.3 Usage

C and Go can be used together in the same address space, but this is not a natural fit and may require some special interface software. Another thing is that, linking C with Go code gives up the memory safety and stack management features that Go provides. There may be cases that it's absolutely necessary to use C libraries to solve a problem, but doing so always introduces some risk not present with pure Go code, which requires special care. If the C and Go combination is absolutely necessary, then the process followed depends on the Go compiler implementation. There are three Go compiler implementations developed by the Go team. These are: `gc`, the default compiler, `gccgo`, which uses the GCC back-end, and a less mature `gollvm` that uses the LLVM infrastructure [1].

The Go project does not include a dedicated IDE. Although, the language itself and the libraries have been designed to make it easy to analyze source code. That means, that most well-known editors and IDEs support Go, either directly or through a plugin. There are some well-known IDEs and text editors that have good Go support available, like Emacs, Vim, Visual Studio Code, Atom, Eclipse, Sublime and Goland [1].

Go now is used worldwide by millions of programmers, whose numbers are growing every day. A lot of companies and projects have made their own success stories using Go, especially in the cloud computing space, but by no means exclusively in this field. For example, Docker and Kubernetes are a couple of major cloud infrastructure projects written in Go [1]. But cloud is not the only reason to choose Go. It can also be used for building command-line interfaces, for classic web development or it can even support DevOps and site reliability [2].

Programmers who want to know more about the Go language, can find a lot of useful resources in the golang.org, go.dev and github.com/golang/go/wiki web pages. These resources include news about the language, download links, extensive packages documentation, essential learning material, social media, a wiki and blog.

1.2 Writing Go code

Go has many great features as a programming language. To start writing Go code, one should have a basic knowledge of the language's most basics aspects. In the following paragraphs, the basic features of Go are going to be demonstrated by providing on-point code examples with thorough explanation.

1.2.1 Packages, variables and functions

1.2.1.1 Packages

Go programs are organized into the so-called "packages". A `package` is a collection of Go source files in the same directory that are compiled together. Types, variables, constants and functions defined in one source file are visible to all other source files within the same package [3]. Packages are imported via their import path, which is the directory path inside the `src` folder of the `GOPATH`. This rule does not apply to the packages of the standard library that can be imported just by specifying their name.

`packages.go` [3]

```go
package main

import "fmt"

func main() {
   fmt.Println("Hello, World!")
}
```

If 2 or more packages must be imported, multiple import statements can also be used. But the best practice is to group the imports into a parenthesized import statement [4].

`imports.go` [5]

```go
package main

import (
   "fmt"
   "math/rand"
)

func main() {
   fmt.Println("Just a random number: ", rand.Intn(10))
}
```

1.2.1.2 Comments

Go has C-style `/* */` block comments and C++-style `//` line comments. Line comments are more common than block comments, as they appear mostly as package comments, but are useful within an expression or to disable large chunks of code. [6].

`comments.go` [3]

```
/*
This is a block comment that describes the package.
It contains multi-line comments.
 */
package main

import "fmt"

// This is a single-line comment that provides the function's description.
func main() {
   fmt.Println("Hello, World!")
}
```

1.2.1.3 Exported names

A name in Go is exported only if it begins with a capital letter. For example, `Pi` is an exported name, which is exported from the `math` package. `pi` does not start with a capital letter, so it will not be exported.

When importing a package, references can be made only to its exported names. Any "unexported" names are not going to be accessible from outside the package [7].

`exported-names.go` [7]

```
package main

import (
   "fmt"
   "math"
)

func main() {
   // This line will throw an error.
   fmt.Println(math.pi)

   // This line won't, as Pi is an exported name from package math.
   fmt.Println(math.Pi)
}
```

1.2.1.4 Functions

Functions in Go are declared using the `func` keyword. They can take zero or more arguments. The variables' type comes after their name [8]. If 2 or more function parameters are of the same type, it can be omitted from all but the last [9].

Multiple values can also be returned by a function [10]. But these values can be named as well. In such case, they are treated as variables defined at the top of the function. Their names re used to document the meaning of the returned values [11].

A `return` statement without receiving arguments, returns these named values and is known as a “naked” `return`. They should be used only in small functions, as they will harm readability in bigger ones [11].

`functions.go` [8] [9] [10] [11]

```go
package main

import "fmt"

// Function add1 takes 2 arguments of int.
func add1(x int, y int) int {
   return x + y
}

// Function add2 has a shortened parameter declaration
func add2(x, y int) int {
   return x + y
}

// Function swap returns 2 string values
func swap(x, y string) (string, string) {
   return y, x
}

// Function split performs a naked return of 2 named values
func split(sum int) (x, y int) {
   x = sum * 4 / 9
   y = sum - x
   return
}

func main() {
   fmt.Println(add1(2,3))
   fmt.Println(add2(3,4))

   s1, s2 := swap("first", "second")
   fmt.Println(s1, s2)

   fmt.Println(split(15))
}
```

1.2.1.5 Variables

The `var` statement is used to declare variables, either multiple or not, either of the same type or different. This statement can be at package or function level [12]. Also it can include initializers, one per variable. So if an initializer is present, there is no need to declare the variable’s type, as it will take the initializer’s type [13].

While in function scope and only, the `:=` short assignment can be used instead of the `var` statement [14].

`variables.go` [12] [13] [14]

```go
package main

import "fmt"

// variables declared at package level
var i, j, k bool

// variables at package level with initializers
var(
   a, b = 1, 2
   c    = true
)

func main() {
   // variable declared at function level
   var l int
   fmt.Println(i, j, k, l)

   // variable at function level with initializer
   var d = "hi"
   // variable created with short declaration
   e := 1.5
   fmt.Println(c, d, e)
}
```

1.2.1.6 Basic Types

Go's basic types are the following: `bool`, `string`, `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `uintptr`, `byte`, `rune`, `float32`, `float64`, `complex64`, `complex128` [15]. But when a variable is declared without an initial value, it is given its zero value. Zero values are `0` for numeric types, `false` for the boolean type and “” empty string for strings [16].

Type conversions can also be performed in Go, using the expression `T(v)` that converts the value `v` to the type `T`. For example, a variable `i` declared as `int` with value `5`, can be converted to `float64` by typing `float64(i)` [17].

It should be noted that when a variable is declared without specifying a type when using `:=` or `var =`, the variable's type is inferred from the value on the right hand side of the assignment [18].

`basic-types.go` [15] [16] [17] [18]

```go
package main

import (
   "fmt"
   "math"
   "math/cmplx"
)

var (
   // Declaration of several types of variables
   WhatIs  bool       = false
   HugeInt uint64     = 1<<64 - 1
   z       complex128 = cmplx.Sqrt(-5 + 12i)
)
```

```
var (
   // Declaration of variables with zero values
   anInt     int
   aFloat64 float64
   aBool     bool
   aString  string
)

var (
   //  Convert some variables' types
   cx, cy int      = 3, 4
   cf     float64 = math.Sqrt(float64(cx*cx + cy*cy))
   cz     uint     = uint(cf)
)

func main() {
   fmt.Printf("Type: %T Value: %v\n", WhatIs, WhatIs)
   fmt.Printf("Type: %T Value: %v\n", HugeInt, HugeInt)
   fmt.Printf("Type: %T Value: %v\n", z, z)

   fmt.Printf("%v %v %v %q\n", anInt, aFloat64, aBool, aString)

   fmt.Println(cx, cy, cz)

   // The variable's type is complex128 without specifying it directly
   complexNum := 5 + 2i
   fmt.Printf("v is of type %T\n", complexNum)
}
```

1.2.1.7 Constants

Constants in Go are just like variables, but they are declared using the `const` keyword and they can be characters, strings, booleans or numerics. They cannot be declared using the short assignment := [19]. It should be noted that numeric constants are high-precision values, but also untyped constants take the type needed by the current context [20].

`constants.go` [19] [20]

```
package main

import "fmt"

// Constant declaration in package scope
const E = 2.718

const (
   // Numeric constants declaration
   bigInt   = 1 << 100
   smallInt = bigInt >> 99
)

func getInt(x int) int {
   return x * 10 + 1
}

func getFloat(x float64) float64 {
   return x * 0.1
}

func main() {
   // Constants declaration in function scope
   const World = "World!"
   const ItIs = true
```

```
    fmt.Println("Hello,", world)
    fmt.Println("e =", E, "is the base of natural logarithms.")
    fmt.Println("Go is the best:", ItIs)

    // Untyped constants take the type of the needed context
    fmt.Println(getInt(smallInt))
    fmt.Println(getFloat(smallInt))
    fmt.Println(getFloat(bigInt))
}
```

1.2.2 Flow control statements

1.2.2.1 For

The only looping construct in Go is the `for` loop. It has 3 basic components: the initialization statement which is executed before the first iteration, the condition expression that is evaluated before every iteration and the post-iteration statement executed after the end of every iteration. The initialization statement is usually a short variable declaration, visible only inside the `for` loop's scope. The iterations stop when the condition evaluates to `false` [21].

The initialization and post-iteration statements are optional [22]. If both are dropped, this is the equivalent `while` of other languages in Go [23]. Also without the condition, the loop loops forever [24].

for.go [21] [22] [25]

```
package main

import "fmt"

func main() {
    // For loop with 3 components
    for a := 0; a < 3; a++ {
        if a == 1 {
            continue
        }
        fmt.Println(a)
    }

    // For loop without the post-iteration statement
    for b := 0; b < 3; {
        fmt.Println(b)
        b++
    }

    // For loop only with condition (while loop)
    c := 0
    for c < 3 {
        fmt.Println(c)
        c++
    }

    // Infinite for loop
    for {
        fmt.Println("just loopin'")
        break
    }
}
```

1.2.2.2 If

Go's `if` statement evaluates certain conditions without needing parentheses, only curly braces just like the `for` loop [26]. It can start with a short statement that is executed before the condition. Any variable declared by the statement belongs only in the `if` statement scope [27]. As a result, it is also available inside `else if` and `else` blocks [28].

`if.go`

```go
package main

import "fmt"

func main() {
   // Normal if statement
   num := 3
   if num == 3 {
      fmt.Println("num is 3")
   }

   // If statement with initial short statement
   for i := 0; i < 3; i++ {
      if j := 1; j < i {
         fmt.Println("Inside the if statement!")
      }
      fmt.Println("Inside the for loop!")
   }

   // If, else if and else blocks
   if v := 2; v == num {
      fmt.Println("v is equal to num")
   } else if v < num {
      fmt.Println("v is smaller than num")
   } else {
      fmt.Println("v is greater than num")
   }
}
```

1.2.2.3 Switch

Instead of writing a sequence of `if - else` statements, a `switch` statement could be a better alternative. Unlike other languages like C, C++ or Java, `switch` runs only the selected case and not all the cases that follow, meaning that the `break` statement at the end of each case needed in other languages is not needed in Go. Also the `switch` cases don't have to be constants and the values involved need not be integers [29].

The `switch` cases evaluate from to top to bottom and if a case succeeds, the case evaluation stops [30]. A `switch` condition is not always needed. Omitting it, it is the same as writing `switch true`, a technique that provides cleaner implementation of `if - else` chains [31].

`switch.go` [29] [30] [31]

```go
package main

import (
   "fmt"
   "runtime"
   "time"
)

func main() {
   // Switch statement. Notice the initial short variable declaration
   fmt.Print("My OS is: ")
   switch os := runtime.GOOS; os {
      case "darwin":
         fmt.Println("OS X")
      case "linux":
         fmt.Println("Linux")
      default:
         fmt.Printf("%s.\n", os)
   }

   // Switch statement with cases that are variables
   fmt.Print("When's June? ")
   month := time.Now().Month()
   switch time.June {
      case month + 1:
         fmt.Println("In 1 month")
      case month + 2:
         fmt.Println("In 2 months")
      case month + 3:
         fmt.Println("In 3 months")
      default:
         fmt.Println("Too far away...")
   }

   // Switch statement without condition
   current_time := time.Now()
   switch {
      case current_time.Hour() < 12:
         fmt.Println("Good morning")
      case current_time.Hour() < 17:
         fmt.Println("Good afternoon")
      default:
         fmt.Println("Good evening")
   }
}
```

1.2.2.4 Defer

The `defer` statement delays the execution of a called function until the function that contains it returns, although the deferred call's arguments are evaluated immediately [32]. All the deferred function calls are pushed onto a stack, so when a function returns, they are executed in LIFO order [33].

Using `defer`, is an unusual but effective way to tackle situations where certain resources must be released, such as closing a file or releasing a mutex, regardless of which path a function takes to return [6].

`defer.go` [32] [33]

```go
package main

import "fmt"
```

```go
// Function defer1 shows a simple defer functionality
func defer1() {
   defer fmt.Println("What's up ?")

   fmt.Print("Hey! ")
}

// Function defer2 demonstrates the LIFO execution of the deferred calls
func defer2() {
   fmt.Println("Starting to count:")

   for i := 0; i < 5; i++ {
      defer fmt.Printf("defer inside loop, i=%v\n", i)
   }
   defer fmt.Println("defer outside of loop")

   fmt.Println("Done counting")
}

func main() {
   defer1()
   defer2()
}
```

1.2.3 Structs, slices and maps

1.2.3.1 Pointers

Like C, Go has pointers, but no pointer arithmetic. Pointers hold the memory address of a value. A pointer to a certain type `T` has the `*T` type and if it doesn't have a value, its value is `nil` [34].

Pointers use 2 operators. The & operator creates a pointer to its operand, while the * operator denotes the pointer's underlying value [34].

`pointers.go` [34]

```go
package main

import "fmt"

func main() {
   // Declare two integer variables
   var1, var2 := 4, "hi"

   // Create two pointers, each for every variable
   var p1 *int
   p1  = &var1
   p2 := &var2

   // Read the variables through their pointers
   fmt.Println("Reading var1 value through p1: ", *p1)
   fmt.Println("Reading var2 value through p2: ", *p2)

   // Set the variables' values through the pointers
   *p1 = *p1 + 4
   *p2 = "ha"
   fmt.Println("Changed var1 value: ", *p1)
   fmt.Println("Changed var2 value: ", *p2)
}
```

1.2.3.2 Structs

A `struct` in Go is a collection of fields of various types [35]. Those fields can be accessed via a dot notation [36]. They can also be accessed through a struct pointer. For a field `X` to be accessed with the struct pointer `p`, the use of the `(*p).X` statement would be ideal. But Go permits just the use of `p.X`, without the explicit dereference [37].

Struct literals declare newly allocated structs by defining the values of their fields. A subset of those fields can be listed using the `Field:` syntax as shown below. The order of the fields does not matter [38].

`structs.go` [35] [36] [37] [38]

```go
package main

import "fmt"

// Create a simple struct
type Point struct {
   X int
   Y int
}

func main() {
   // Create and print a Point struct with specific X and Y
   point1 := Point{1, 2}
   fmt.Println(point1)

   // Gain access to a Point's X field
   point2  := Point{3, 4}
   point2.X = 4
   fmt.Println(point2.X)

   // Access a struct's field with a pointer to that struct
   point3 := Point{4, 5}
   p3     := &point3
   p3.X    = 1e6           //(*p3).X will also do
   fmt.Println(point3)

   // Create structs with various struct literals
   point4 := Point{5, 6}  // has type Point
   point5 := Point{X: 7}  // Y:7 is implicit
   point6 := Point{}      // X:0 and Y:0
   p      := &Point{8, 9} // has type *Point
   fmt.Println(point4, point5, point6, p)
}
```

1.2.3.3 Arrays

Arrays in Go are declared as `[n]T`, where `n` is the number of the array's elements and `T` the type of the array. Each array's length is part of its type, so arrays can't be resized [39].

`arrays.go` [39]

```go
package main

import "fmt"

func main() {
   // Create an empty array of fixed size, then assign values to the elements
   var arr [3]string
   arr[0] = "Array"
   arr[1] = "Of"
   arr[2] = "Strings"
   fmt.Println(arr[0], arr[1], arr[2])
   fmt.Println(arr)

   // Create an array with values
   comp := [6]int{4, 6, 8, 9, 10, 12}
   fmt.Println(comp)
}
```

1.2.3.4 Slices

Slices are dynamically-sized, flexible views into the elements of an array and they are more common than arrays in practice. The `[]T` type is a slice that contains elements of type `T`. A slice is created by selecting a low and a high bound of an array, thus specifying a half-open range that includes the first element, but excludes the last one [40]. As a result, a slice does not store data, but it describes a section of the underlying array. If the elements of the slice are changed, the corresponding elements of its underlying array are changed as well. If more than one slices share the underlying array, they see those changes too [41].

Slice literals are like array literals but without the length [42]. When slicing, the high and low bound can be omitted in order to use the default ones. It is 0 for the lower bound and the slice's length for the higher bound [43].

A slice has a length and a capacity. Length is the number of its elements and capacity is the number of elements in the underlying array, counting from the first element in the slice. A slice's length and capacity can be directly obtained using the built-in expressions `len(slice)` and `cap(slice)`. The length can be extended by performing a re-slicing, but only if there is enough capacity [44].

The slices' 0 value is `nil`. A `nil` slice does not have an underlying array and its length and capacity are 0 [45]. Slices can also be created with the built-in function `make([]T, length, capacity)`, making it possible to create dynamically sized arrays. [46]. They can contain any type of data, as well as other slices [47]. Also slices can receive new elements. This can be done with the built-in `append([]T, elem1, elem2…)` function, which takes in a slice, appends new elements to it and returns the new slice [48].

Slices and maps can be iterated with the `range` form of the `for` loop. When inside a `range` loop, 2 values are returned for each iteration, the current slice index and a copy of the element at the current index [49]. The index or the element's copy can be omitted by assigning to _. If the element's copy is to be omitted, it does not have to be assigned to _, but it can be completely ignored [50].

slices.go [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50]

```
package main

import (
   "fmt"
   "strings"
)

func main() {
   // First create an array
   primes := [8]int{0, 1, 2, 3, 4, 5, 6, 7}
   // Then create a slice that contains the elements 1 to 4 of the array
   var sl1 []int = primes[1:5]
   fmt.Println("Slice sl1 of the primes array:", sl1)

   // Create slices of an array, changes values and notice the changes in both
slices and the array
   nums := [5]string{"one", "two", "three", "four", "five"}
   fmt.Println("The nums array:", nums)
   sl2 := nums[0:3]
   sl3 := nums[1:4]
   fmt.Println("A slice sl2 of the nums array:", sl2)
   fmt.Println("Another slice sl3 of the nums array:", sl3)
   sl3[1] = "CHANGE"
   fmt.Println("Slice sl2 changed:", sl2)
   fmt.Println("Slice sl3 changed:", sl3)
   fmt.Println("nums array changed:", nums)

   // Slice literals
   sl4 := []bool{true, false, true, true, false, true}
   fmt.Println("Slice sl4:", sl4)

   // Slice defaults
   sl5 := sl4[:2]
   sl6 := sl4[2:5]
   sl7 := sl4[4:]
   fmt.Println("Slice sl4 defaults:", sl5, sl6, sl7)

   // Slices' length and capacity
   sl8 := []int{1, 3, 5, 7, 9, 11}
   fmt.Printf("The sl8 slice: %v len=%d cap=%d\n", sl8, len(sl8), cap(sl8))
   // Slice the slice to give it zero length
   sl8 = sl8[:0]
   fmt.Printf("sl8 re-sliced 1st time: %v len=%d cap=%d\n", sl8, len(sl8),
cap(sl8))
   // Extend its length
   sl8 = sl8[:4]
   fmt.Printf("sl8 re-sliced 2nd time: %v len=%d cap=%d\n", sl8, len(sl8),
cap(sl8))
   // Drop its first two values
   sl8 = sl8[2:]
   fmt.Printf("sl8 re-sliced 3rd time: %v len=%d cap=%d\n", sl8, len(sl8),
cap(sl8))

   // A nil slice
   var sl9 []int
   fmt.Printf("The sl9 slice: %v len=%d cap=%d\n", sl9, len(sl9), cap(sl9))
   if sl9 == nil {
      fmt.Println("sl9 is nil")
   }

   // Create slice with make
   sl10 := make([]int, 0, 6)
```

```go
   fmt.Printf("The sl10 slice: %v len=%d cap=%d\n", sl10, len(sl10), cap(sl10))
   sl11 := sl10[:2]
   fmt.Printf("sl11 slice created by sl10: %v len=%d cap=%d\n", sl11,
len(sl11), cap(sl11))

   // Slices of slices
   // Create a tic-tac-toe board.
   board := [][]string{
      []string{"_", "_", "_"},
      []string{"_", "_", "_"},
      []string{"_", "_", "_"},
   }
   // The players take turns.
   board[0][0] = "O"
   board[2][2] = "X"
   board[1][2] = "X"
   board[1][0] = "O"
   board[0][2] = "X"
   for i := 0; i < len(board); i++ {
      fmt.Printf("%s\n", strings.Join(board[i], " "))
   }

   // Append new elements to a slice
   var sl12 []string
   fmt.Printf("The sl12 slice: %v len=%d cap=%d\n", sl12, len(sl12), cap(sl12))
   sl12 = append(sl12, "hi")
   sl12 = append(sl12, "there", "!")
   fmt.Printf("The sl12 slice with new elements: %v len=%d cap=%d\n", sl12,
len(sl12), cap(sl12))

   // Range loop a slice
   sl13 := []float64{0.1, 0.2, 0.3, 0.4}
   for i, v := range sl13 {
      fmt.Printf("Element in index %d has value %v\n", i, v)
   }

   // Range without value
   for i := range sl13 { // can also be written: for i, _ := range...
      fmt.Printf("Element in index %d\n", i)
   }
   // Range without index
   for _, v := range sl13 {
      fmt.Printf("Element has value %v\n", v)
   }
}
```

1.2.3.5 Maps

Maps map keys to values. A new map can be created with the `make` function. A map can have `nil` value, just like slices [51]. Map literals are just like struct literal with the extra need to specify the keys [52]. Data mutations can also take place inside maps. These mutations include the insertion or the update of a map's element and the retrieval or the deletion of an element. Besides these functions, a test that a specific key is present inside the map can be done with a two-value assignment [53]. In addition, maps can be iterated with the `range` form of the `for` loop.

`maps.go` [51] [52] [53]

```go
package main

import "fmt"
```

```go
type Point struct {
   X, Y float64
}

func main() {
   // Create a new map and give a value to it
   m1 := make(map[string]Point)
   m1["1st Point"] = Point{13.859, 20.136}
   fmt.Println("1st Map's 1st Point:", m1["1st Point"])

   // Create a map with map literals
   m2 := map[string]Point{
      "1st Point": {10.500, 11.500},
      "2nd Point": {12.600, 13.600},
      "3rd Point": {15.800, 16.800},
   }
   fmt.Println("2nd Map:", m2)

   // Mutate a map
   m3 := make(map[string]float64)
   // Insert an element
   m3["1st Point"] = 13.333
   fmt.Println("Inserted an element in map m3:", m3)
   // Update an element
   m3["1st Point"] = 14.444
   fmt.Println("Updated an element in map m3:", m3)
   // Delete an element
   delete(m3, "1st Point")
   fmt.Println("Deleted an element in map m3:", m3)
   // Check if a key exists
   value, ok := m3["1st Point"]
   fmt.Println("The key '1st Point' with value", value, "exists:", ok)

   // Iterate a map with range
   fmt.Println("Iterating the m2 map...")
   for i, v := range m2 {
      fmt.Printf("Element with key %s has value %v\n", i, v)
   }
}
```

1.2.3.6 Function values and closures

In Go, functions are values too and they can be passed around like the other values such as `int`. For example, they can be used as function arguments and return values [54].

Go functions can also be closures. As functions can be values too, closures are function values that reference variables from outside their bodies. This function can access and set those referenced variables [55].

`function-values-closures.go` [54] [55]

```go
package main

import (
   "fmt"
   "math"
)

// A function that takes a function as an argument.
// It also returns that function
func compute(fn func(float64, float64) float64) float64 {
   return fn(3, 4)
}
```

```go
// The adder function returns a closure that is
// bound to its own sum variable
func adder() func(int) int {
   sum := 0
   return func(x int) int {
      sum = sum + x
      return sum
   }
}

func main() {
   // Create a new function and save it to a variable
   hypot := func(x, y float64) float64 {
      return math.Sqrt(x*x + y*y)
   }
   // Use this function directly
   fmt.Println(hypot(3, 5))
   // Use the function as argument in the compute function
   fmt.Println(compute(hypot))
   // math.Pow used as argument in the compute function.
   // It is exactly the same type as hypot, but it returns
   // the base-x exponential of y
   fmt.Println(compute(math.Pow))

   // Use the closure the adder function returns
   closure := adder()
   for i := 0; i < 5; i++ {
      fmt.Println("Sum is:", closure(i))
   }
}
```

1.2.4 Methods and interfaces

1.2.4.1 Methods

Go doesn't support classes, but methods can be defined in types. Methods are nothing but functions with a special receiver argument that appears between the `func` keyword and the method's name [56]. A method can be declared on non-struct types too. Methods can only be declared on types that are defined in the same package as the method. If the type is defined in another package, a method on that type cannot be declared [58].

Methods can have pointer receivers, meaning that the receiver is of `*T` type, without `T` already being a pointer. A method with a pointer receiver can modify the value to which the receiver points. As a result, pointer receivers are more common than value receivers, since methods often need to modify their receiver. Functions have also the same behavior. If a function takes a pointer argument, then it can modify the value the pointer points to. If it takes a value argument, it cannot [59].

Go offers a great convenience: methods with pointer receivers take either a value or a pointer as a receiver when they are called. However, that doesn't happen with normal functions, as they should take a pointer argument, if that's their specification [61]. In the reverse direction, methods with value receivers take either a value or a pointer when they are called, while functions that take value arguments, should be given only value arguments [62].

Pointer receivers for methods are better to use for two main reasons. First, with pointer receivers, the method can modify the value the pointer points to. Second is to avoid copying the value on each method call, which is really useful is the receiver is a large struct for example [63].

`methods.go` [56] [57] [58] [59] [60] [61] [62]

```go
package main

import (
   "fmt"
   "math"
)

type Point struct {
   X, Y float64
}

// Declare an Abs method for the Point struct.
// Abs has a value receiver, which means that it operates
// on a copy of the original Point value
func (p Point) Abs() float64 {
   return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

// The Scale method has a pointer receiver, because it
// should be able to change the Point's value that is called upon
func (p *Point) Scale(f float64) {
   p.X = p.X * f
   p.Y = p.Y * f
}

// The Abs method written as a normal function
// that takes a Point as an argument
func Abs(p Point) float64 {
   return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

// The Scale method written as normal function, with a
// pointer to a Point struct as argument
func Scale(p *Point, f float64) {
   p.X = p.X * f
   p.Y = p.Y * f
}

type MyInt int

// Declare an Absolute method for the non-struct type MyInt
func (i MyInt) Absolute() int {
   if i < 0 {
      return int(-i)
   }
   return int(i)
}

func main() {
   // Call the Abs method of a Point struct
   p1 := Point{3, 4}
   fmt.Println("p1 Abs method:", p1.Abs())

   // Call the Abs function that takes a Point as an argument
   fmt.Println("Abs function with p1 as argument:", Abs(p1))
```

```
    // Call the Absolute method of a MyInt variable
    mi1 := MyInt(-5)
    fmt.Println("mi1 Absolute method:", mi1.Absolute())

    // Call the Scale method to change a Point
    p2 := Point{1, 2}
    fmt.Println("p2 before Scale method:", p2)
    // The Scale method has a pointer receiver, but Go
    // interprets p2.Scale as (&p2).Scale
    p2.Scale(2) // (&p2).Scale will produce the same result
    fmt.Println("p2 after Scale method:", p2)

    // Call the Abs and Scale functions instead of the methods
    p3 := Point{0.5, 0.5}
    Scale(&p3, 2)
    fmt.Println("Scale function with a pointer to p3 as argument:", p3)
    fmt.Println("Abs function with p3 as argument:", Abs(p3))

    // p4 is a pointer, but Abs method can be called
    // although it has a value receiver.
    // Abs function needs a value argument, doesn't receive a pointer one
    p4 := &Point{5, 6}
    fmt.Println("p4 Abs method:", p4.Abs())
    fmt.Println(("Abs function with dereferencing on p4 pointer to struct:"),
Abs(*p4))
}
```

1.2.4.2 Interfaces

An interface is a type itself, and is defined as a set of method signatures. An interface type value can be of any value, as long as this value is of a type that implements these methods [64]. Simply put, a type implements an interface by implementing its methods. There is no need for explicit declaration of such intent [65].

Interface values can be conceived as a tuple of a value and a solid type, like `(value, type)`. The interface's value holds a value of a specific underlying solid type. If a method is called on an interface value, the same method of the underlying type is executed [66]. The value of the interface can also be `nil`. Should that be the case, the method will be called with a `nil` receiver, without causing any exception. An interface that has a `nil` value is itself non-nil [67]. An interface can be completely `nil`, meaning that it has neither a value nor a solid type. Methods can't be called in `nil` interfaces, as they don't have a specific type to indicate which method to call [68].

There is an interface type that specified zero methods. It is called the empty interface, `interface{}`. Empty interfaces can hold values of any type and are used by code that handles of unknown type [69].

Go also supports type assertions. They provide access to an interface's value and underlying type. The type assertion statement is `t, ok := i.(T)`, which is an assertion that the interface `i` holds the type `T`. A type assertion returns two values. The first is the interface's value and the second is a boolean value that states if the assertion was successful. The boolean value can be omitted, so the type assertion can also be written `t := i.(T)`. In this statement, if `i` is not of the `T` type then a panic is triggered [70].

Go also has a convenient way to perform several type assertions in series. A type switch is just like the normal `switch` statement, but the cases here are about types. The type values specified by those cases are compared against the type of the given interface that is being switched [71].

interfaces.go [64] [66] [67] [68] [69] [70] [71]

```go
package main

import "fmt"

// An interface that has a single method, Invert
type Inverter interface {
   Invert()
}

// Type TwoDPoint implements the Inverter interface
type TwoDPoint struct {
   X, Y float64
}

// Implicit implementation of the Inverter interface
func (tdp *TwoDPoint) Invert() {
   tdp.X = -tdp.X
   tdp.Y = -tdp.Y
}

// Type OneDPoint implements the Inverter interface
type OneDPoint struct {
   X float64
}

// Implicit implementation of the Inverter interface
func (odp *OneDPoint) Invert() {
   odp.X = -odp.X
}

func main() {
   // Create Inverter, TwoDPoint and OneDPoint variables
   var i1, i2 Inverter
   p1 := TwoDPoint{1, 2}
   p2 := OneDPoint{3}
   i1 = &p1 // Only *TwoDPoint implements Inverter, not TwoDPoint
   i2 = &p2 // Only *OneDPoint implements Inverter, not OneDPoint
   i1.Invert()
   i2.Invert()
   fmt.Println("*TwoDPoint type var p1 implements Inverter:", i1)
   fmt.Println("*OneDPoint type var mf1 implements Inverter:", i2)

   // Check interface values and types
   fmt.Printf("Value and type of i1 interface: (%v, %T)\n", i1, i1)
   fmt.Printf("Value and type of i2 interface: (%v, %T)\n", i2, i2)

   // Interface with nil value
   var (
      i3 Inverter
      p3 *TwoDPoint
   )
   i3 = p3
   fmt.Printf("Value and type of i3 interface: (%v, %T)\n", i3, i3)

   // Nil interface
```

```
    var i4 Inverter
    fmt.Printf("Value and type of i4 interface: (%v, %T)\n", i4, i4)
    // Calling Invert on i4 causes run-time error: i4 doesn't have a type
    // i4.Invert()

    // Empty interface
    var i5 interface{}
    fmt.Printf("Value and type of i5 interface: (%v, %T)\n", i5, i5)
    i5 = "hi"
    fmt.Printf("Value and type of i5 interface: (%v, %T)\n", i5, i5)
    i5 = 5
    fmt.Printf("Value and type of i5 interface: (%v, %T)\n", i5, i5)

    // Type assertions
    var i6 interface{} = 88.88
    t, ok := i6.(float64) // ok can be omitted, can be written t := i6.(float64)
    if ok {
        fmt.Printf("Type assertion passed: i6 (%v, %T)\n", t, i6)
    }
    // The following line triggers a panic
    // t1 := i6.(string)

    // Type switch
    switch v := i6.(type) {
        case int:
            fmt.Printf("INT! i6 is of type %T with value %v\n", v, v)
        case float64:
            fmt.Printf("FLOAT64! i6 is of type %T with value %v\n", v, v)
        default:
            fmt.Printf("DEFAULT CASE! i6 is of type %T!\n", v)
    }
}
```

1.2.5 Concurrency

1.2.5.1 Goroutines

Goroutines are lightweight threads managed by the Go runtime. A goroutine is initiated with the syntax `go function(args)`. The evaluation of `args` happens in the main goroutine and the execution of function in the new goroutine [72]. In the following example, the `time.sleep(10 * time.Millisecond)` is used to delay the termination of the `main` goroutine and schedule the execution of the `printSomething` goroutine, otherwise the latter will not be executed at all [73].

goroutines.go [72] [73]

```
package main

import (
    "fmt"
    "time"
)

// Function printSomething will run as a goroutine
func printSomething() {
    fmt.Println("The printSomething goroutine is executed")
}

func main() {
    fmt.Println("The main goroutine started")

    // Initialize a goroutine
    go printSomething()
    // Delay the termination of main goroutine to schedule the
    // execution of the printSomething goroutine
    time.Sleep(10 * time.Millisecond)

    fmt.Println("The main goroutine ended")
}
```

1.2.5.2 Channels

Maybe Go’s most popular data structures are channels. A channel is actually something like a pipe, through which values can be sent and received using the channel operator `<-`. Data flows in the direction of the arrow; `ch <- v` sends value `v` to the channel and `v := <-ch` receives a value from `ch` and assigns it to `v`. Before use, channels should be first created with the `make` function. The default behavior of channels is that, the one side of the channel is blocked from sending or receiving until the other side is ready to send or receive. That makes goroutines easy to synchronize without using locks or condition variables [74]. Channels can also be buffered, so that they only hold a limited number of values. The buffer length is provided as the second argument inside the make function like this, `ch := make(chan int, buffer_size)`. If the buffer is full, the channel blocks from sending new values to it, but if the buffer is empty, the channel blocks from receiving values from it [75].

Channels can be looped with the `range` loop to receive values repeatedly from the channel until it is closed. The syntax is `for i := range ch`. The side of a channel that sends values can `close` that channel, in order to stop sending values into it. Only the sender side of a channel can close it, not the receiver one. Sending a value on a closed channel is going to cause a panic. But channels don’t usually need to be manually closed, it’s only necessary when the receiver side must be notified that there will be no more values coming. One such case would be to terminate a `range` loop. Also the receiver side can test whether the channel has been closed. This can be done by assigning a second parameter in the channel receive expression: `v, ok := <-ch`. If there are no more values to receive, ok becomes `false` [76].

A goroutine can wait on multiple communication operations with the `select` statement. Using `select`, the execution of the goroutine is blocked until one of the cases can be run. If multiple conditions are ready to be executed, one of them is selected randomly [77]. If the `default` case is used inside the `select` statement, it is executed if no other case is ready [78].

channels.go [74] [75] [76] [77] [78]

```go
package main

import (
   "fmt"
   "time"
)

// Function sum sums all the elements of the input slice,
// then sends the computed sum value inside a channel that
// receives integer values
func sum(sl []int, ch chan int) {
   sum := 0
   for _, v := range sl {
      sum += v
   }
   // Wait till ch is ready to receive values, then send the sum
   // into ch
   ch <- sum
}

// Function fibonacci1 is used to demonstrate the combination
// of close - range functionalities. n is the number of values
// the receiver will request (the capacity of channel ch3)
func fibonacci1(n int, ch chan int) {
   x, y := 0, 1
   for i := 0; i < n; i++ {
      // Send a value to the channel
      ch <- x
      x, y = y, x+y
   }
   // Close the channel after sending all the values to be sent
   close(ch)
}

// Function fibonacci2 is used to showcase the simple
// select with multiple case. It takes two integer channels as arguments
func fibonacci2(ch, quit chan int) {
   x, y := 0, 1
   // The cases evaluation inside the select statement
   // should always be happening, so the select-cases
   // are surrounded by an infinite for loop
   for {
      select {
         // If ch is ready to send values inside it, execute this case
         case ch <- x:
            x, y = y, x+y
         // If quit is ready to receive values, execute this case
         case <-quit:
            return
         // If no other case is ready, the default case is executed
         default:
            fmt.Println("Inside the default case!")
            time.Sleep(100 * time.Millisecond)
      }
   }
}

func main() {
   // Simple demonstration of channel functionality
   // Create a slice and a channel
   sl1 := []int{1, 2, 3, 4, 5, 6}
   ch1 := make(chan int)
   // Create 2 goroutines, where each one computes
   // the sum of the half of the sl1 slice
   go sum(sl1[:len(sl1)/2], ch1)
   go sum(sl1[len(sl1)/2:], ch1)
   // Receive and assign the computed sums into 2 variables.
   // We don't know which goroutine will finish first,
   // so x is not necessarily going to be 6 and y 15
   sum1, sum2 := <-ch1, <-ch1
   fmt.Printf(
      "One sum of sl1 slice: %v\n" +
      "The other sum of sl1 slice: %v\n" +
      "Total sum of sl1 slice: %v\n", sum1, sum2, sum1 + sum2,
   )

   // Buffered channels
   ch2 := make(chan string, 2)
   ch2 <- "say"
   ch2 <- "something"
   // This will cause a deadlock because buffer is filled
   //ch2 <- 3
   fmt.Println("Retrieving from buffered channel ch2:", <-ch2)
   fmt.Println("Retrieving from buffered channel ch2:", <-ch2)
   // An extra retrieval from an empty buffer will
   // also cause a deadlock
   // fmt.Println(<-ch2)

   // Range - Close demonstration
   ch3 := make(chan int, 10)
   // Initiate a goroutine running fibonacci function
   go fibonacci1(cap(ch3), ch3)
   // Consume all the values inside the ch3 channel with
   // a range loop
   fmt.Printf("A fibonacci sequence with %v values (range-close):\n", cap(ch3))
   for i := range ch3 {
      fmt.Println(i)
   }

   // Select inside a goroutine
   ch4, quit := make(chan int), make(chan int)
   // Create a goroutine with a function literal that
   // receives values from ch4 inside a for loop
   fmt.Println("A fibonacci sequence with 10 values (select):")
   go func() {
      for i := 0; i < 10; i++ {
         fmt.Println(<-ch4)
      }
      // If the loop is completed, signal to the quit channel
      // by sending a value into it, to return the fibonacci2 function
      quit <- 0
   }()
   // Run the fibonacci2 function as a normal function
   fibonacci2(ch4, quit)
}
```

1.2.5.3 Mutexes

Although channels are great for communication between goroutines, there might be some occasions where all that's needed, is to make sure that only one goroutine can access a variable at a time and avoid conflicts. This is called mutual exclusion and the way to achieve it is by using the `Lock` and `Unlock` methods of the `sync.Mutex struct`. The code that needs to be executed with mutual exclusion is surrounded by a `Lock` in the start and an `Unlock` in the end of it [79].

`mutexes.go` [79]

```go
package main

import (
   "fmt"
   "sync"
   "time"
)

// Type SafeCounter is safe to use concurrently
type SafeCounter struct {
   safeMap map[string]int
   mux     sync.Mutex
}

// Method Inc increments the counter for the given key
func (sc *SafeCounter) Inc(key string) {
   // Lock so only one goroutine at a time can access the map
   sc.mux.Lock()
   // Increment the value safely
   sc.safeMap[key]++
   fmt.Println(sc.safeMap[key])
   // Unlock to give access to another goroutine
   sc.mux.Unlock()
   //fmt.Println(sc.Value(key))
}

// Method Value returns the current value of the counter for the given key
func (sc *SafeCounter) Value(key string) int {
   // Lock so only one goroutine at a time can access the map
   sc.mux.Lock()
   // Unlock with defer to ensure that the mutex will be unlocked
   // after the function returns
   defer sc.mux.Unlock()
   return sc.safeMap[key]
}

func main() {
   sc := SafeCounter{safeMap: make(map[string]int)}
   for i := 0; i < 100; i++ {
      go sc.Inc("first_key")
   }
   // Delay the execution of the main goroutine to start
   // scheduling the goroutines that increment the map's value
   time.Sleep(time.Second)
   fmt.Println(
      "The final value of the first_key is:",
      sc.Value("first_key"),
   )
}
```

1.3 A conclusion of the Go language

As seen before, Go borrows ideas from existing languages. However, it has some unusual properties that make effective Go programs different in character from programs written in its relatives. If someone attempts to translate a C++ or Java program into Go in a straightforward way, is unlikely to produce a satisfactory result. Then again, thinking about the problem from a Go perspective could produce a successful, but also quite different program. Simply put, to write Go well, it is important to understand its properties and idioms, let alone to know the established conventions for Go programming, such as naming, formatting, program construction, and so on, so that programs written will be easy for other Go programmers to understand [6].

While Go is a relatively new language (publicly released in 2009), it has taken both the programmers' community and the software businesses by storm in the last years. Go has contributed to success stories in the corporate world [80]. In addition, more than 1300 companies all over the world are currently using Go as their primary language [81]. The Go community continues every day to enrich the language with new libraries for fields such as audio and music, GUI development, database tools, game development, image manipulation, hardware controlling, machine learning, security, code analysis, web development and many more [82].

2 An Overview of Distributed Computing

This chapter is a summary of selected topics from Maarten van Steen's & Andrew Tanenbaum's "Distributed Systems" [83]. Therefore, a general reference to [83] exceeds the whole chapter. Reference to other sources is explicitly noted.

2.1 Introduction

Computer systems were, are and continue to change at an overwhelming pace. Until about 1985 computers were very large and expensive. Also, due to lack of ways to connect them, these computers operated independently from one another. However, starting in the mid-1980s, two advances in technology seemed able to change that situation. The first was the development of powerful microprocessors. At first they were 8-bit machines, but soon 16-, 32-, and 64-bit CPUs became common in the market. Multicore CPUs nowadays are really powerful and cheap compared to the ones that existed 30 or 40 years ago, as well as they offer great help tackling the challenge of adapting and developing programs to exploit parallelism. The second fundamental change was the invention of high-speed computer networks. Local-area networks allow even thousands of machines within a building to be connected in such a way, that small amounts of information can be transferred in a few microseconds. But also larger amounts of data can be moved between machines at rates of billions of bits per second. Wide-area networks allow a vast number of machines (hundreds of millions) all over the world to be connected at speeds varying from tens of thousands to hundreds of millions bps.

Along with the development of increasingly powerful and networked machines, the size of computer systems has become a lot smaller. With the most impressive outcome become the smartphones, they are packed with sensors, lots of memory, and a powerful CPU, so these devices are nothing less than full-fledged computers with networking capabilities of course. Following those footsteps, the so-called plug computers are finding their way to the market. They are small computers, often the size of a power adapter that can be plugged directly into an outlet and offer near-desktop performance.

With now having all these technological advances available, it is pretty easy to put together a computing system composed of a large numbers of networked computers, large or small. The computers are generally geographically dispersed, for they are usually said to form a distributed system. The size of a distributed system may vary from a small number of devices to millions of computers. The interconnection network may be wired, wireless, or a combination of these two. Distributed systems are highly dynamic, meaning that computers can join and leave, while the topology and performance of the underlying network continuously changes.

2.1.1 Definition of a distributed system

A variety of definitions of distributed systems have been given in the literature. It would be sufficient to give a loose characterization: “A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system”. This definition refers to two characteristic of distributed systems.

The first one is that a distributed system is *a collection of other computing elements* (generally referred to as nodes), *each one being able to behave independently from one another*. In practice, nodes are programmed to achieve common goals by exchanging messages with each other. A node can be either a hardware device or a software process. The nodes could be ranging from very big high-performance computers to small plug computers or even smaller devices.

The second is that *users, either people or applications, believe they are dealing with a single coherent system*. This means that the autonomous nodes need to collaborate. But the way this collaboration is established should not be noticed by the end users and have the so-called distribution transparency. So processes, data and control dispersion across a computer network’s nodes should at least appear to be hidden, making the distributed system seem as one coherent entity. In a single coherent system the collection of nodes operated the same as a whole, no matter when, when and how interaction between a user and the system takes place. Note that no assumptions are made concerning the way that nodes are interconnected.

In order to assist the development of distributed applications, distributed systems are organized to have an independent layer of software (middleware) that is logically placed on top of the operating systems of the computers that are part of the system. Each

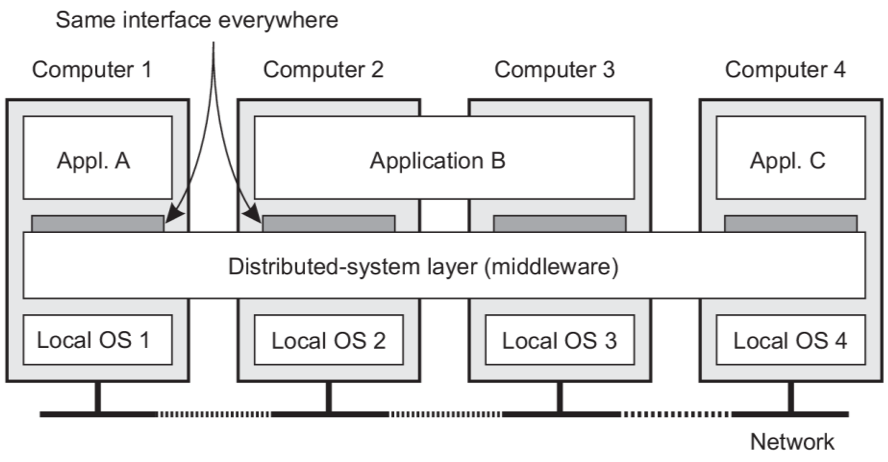


Figure 1. A distributed system organized in a middleware layer

application is offered the same application interface. The distributed system provides the means for the components of a distributed application to communicate with each other, as well as to let different applications communicate. At the same time, it hides, as best as possible, the differences in hardware and operating systems from each application. So middleware can be conceived as a container of commonly used components and functions that they no longer have to be implemented by applications separately. Such middleware is RPC and web services.

2.1.2 Design goals

Just because it is possible to build distributed systems, doesn't really mean that it's actually a good idea. There are four important goals that should be met to make building a distributed system worth the effort.

The first important goal of a distributed system is *to make it easy for users (and applications) to access and share remote resources*. Some typical resources examples include peripherals, storage facilities, data, files, services, and networks. The connection between users and resources makes it easier to collaborate and exchange information worldwide, as shown by the success of the Internet with its protocols for exchanging files, mail, documents, audio, and video. Resource sharing in distributed systems is perhaps best demonstrated by the success of file-sharing peer-to-peer networks that make it extremely simple for users to share files across the Internet like BitTorrent.

The second important goal of a distributed system is to hide that its processes and resources are physically distributed across multiple computers possibly separated by large distances. In other words, it tries *to make the distribution of processes and resources transparent (invisible) to end users and applications*. The concept of transparency can be applied to aspects of distributed systems such as access, location-relocation, migration, replication, concurrency and failure. However, the degree of transparency is a debatable matter that varies depending on the type of each distributed system.

Another goal of distributed systems is *to be open*. An open distributed system is a system that offers components that can easily be used by, or integrated into other systems, while at the same time, it will often consist of components that originate from elsewhere.

In the last years, the number of devices that are being networked has dramatically increased. Even more nowadays, where the cloud applications are dominating. Having this in mind, *scalability* has become one of the most important design goals for developers of distributed systems. Scalability of a distributed system can refer to its size (easy addition of users and resources without performance loss), its geographical distribution (minimal notice of communication delays across large distances) as well as its administration capabilities (easily managed across many independent administrative organizations). These problems can be solved by adopting techniques such as hiding communication latencies, partition and distribution but also replication or caching.

2.2 Communication

Communication between processes is at the heart of all distributed systems. Talking about distributed systems without carefully examining the ways that processes on different machines can exchange information, does not really makes sense. Communication in distributed systems has always been based on low-level message passing as offered by the underlying network. Achieving communication with message passing is harder than using primitives based on shared memory, a technique available only on non-distributed platforms. Thousands or even millions of processes are scattered across a network connecting modern distributed systems with unreliable communication, like the Internet. Development of large-scale distributed applications is extremely difficult, unless the primitive communication facilities of computer networks are replaced by something else.

2.2.1 Foundations

All communication in distributed systems is based on sending and receiving low-level messages because there is no shared memory available. When process P wants to communicate with another process Q, the message is first built in its own address space. Then it triggers a system call that causes the operating system to send the message over the network to Q. In order to prevent chaos, P and Q have to agree on the meaning of the bits being sent, although this basic idea sounds somewhat simple.

To make it easier to deal with the many levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that identifies the various levels involved in a clear way, gives them standard names, and points out which job should be done by which level. The model is called the Open Systems Interconnection Reference Model usually abbreviated as just the OSI model. It is designed to allow open systems to communicate. Open systems are the ones that are prepared to communicate with other opens system by using standard rules that dictate the format, contents, and meaning of the messages sent and received. These rules are formalized in the so-called *communication protocols*. If a group of computers is to communicate over a network, they must all agree on the protocols to be used, where protocols are said to provide communication services. There are two types of these services. In a *connection-oriented service,* the sender and receiver must first explicitly establish a connection, and possibly negotiate specific parameters of the protocol they will use, before exchanging data. When they are done communicating, they terminate the connection between them, just like when two people talk on the telephone. With a *connectionless service,* no setup in advance is needed, the sender just transmits the first message when it is ready. This connection is analogous to dropping a mail inside a mailbox.

In the OSI model, communication is divided into seven layers. Each layer offers specific communication services to the layer above it. This way, the problem of getting a message from the sender to the receiver can be divided into manageable pieces, each able to be solved independently of the others. Every layer provides an interface to the layer above it. The interface consists of a set of operations that together define the service the layer is prepared to offer. The OSI layers are: *Physical layer*: Standardizes how two computers are connected and how 0s and 1s are represented. *Data link layer*: Detects and possibly corrects transmission errors, as well as protocols to keep a sender and receiver in the same pace. *Network layer*: Holds the protocols for routing a message through a computer network, but also the protocols for handling congestion. *Transport layer*: It mainly contains protocols for direct support of applications, such as the ones that establish reliable communication or support real-time streaming of data. *Session layer*: Responsible for handling sessions between applications. *Presentation layer*: Determines the way data is represented so that is independent of the hosts on which communicating applications are running. *Application layer*: Everything else like e-mail protocols, Web access protocols, file-transfer protocols, and many others.

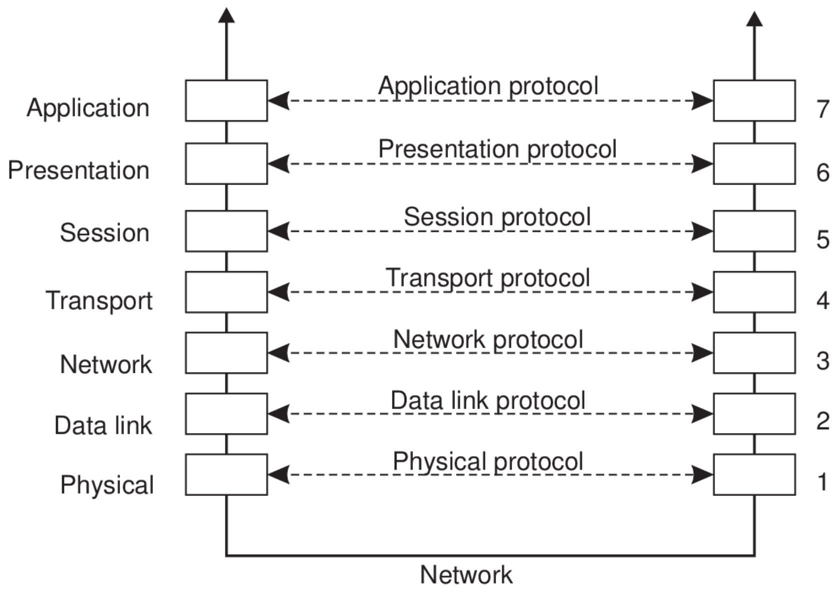


Figure 2. The OSI model's layers, interfaces and protocols

When a process P has to communicate with a remote process Q, at first it builds a message and passes it to the application layer through an interface. Then the application layer software adds a header to the front of the message and passes it across the layer 6/7 interface to the presentation layer, which adds its own header and passes the result down to the session layer and so on. Some layers also add a trailer to the end. When the message hits the physical layer, it transmits the message by putting it onto the physical transmission medium. When the message arrives in the machine hosting Q, it is lifted upwards on each layer where the layer headers are examined respectively.

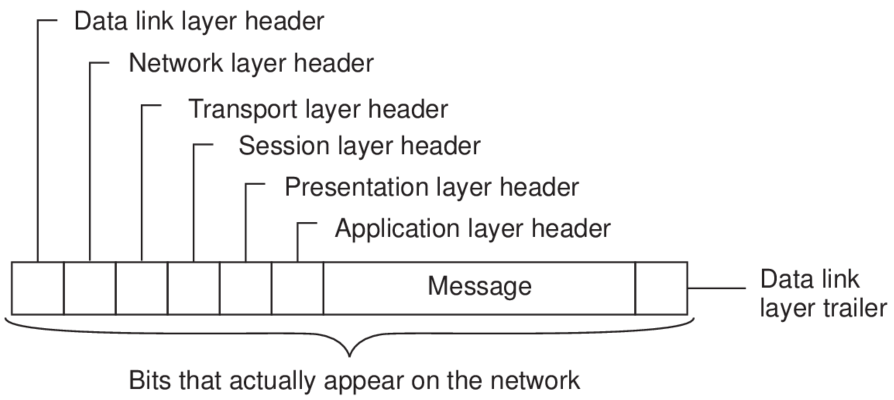


Figure 3. A message as it appears inside the network

The OSI protocols were never popular, in contrast to the protocols developed for the Internet. The most widely used network layer protocol is the connectionless *Internet Protocol (IP)*, where an IP packet (a message when in network layer) can be sent without any setup and is routed to its destination independent from the other packets. The transfer protocol used by the Internet is the *Transmission Control Protocol (TCP)*. Combined with IP (TCP/IP), is nowadays used as a de facto standard for network communication. There is also another transport protocol for the Internet, the connectionless *Universal Datagram Protocol (UDP)*, that is common to the IP, but it is used mainly by programs that don't require a connection-oriented protocol to function properly.

In the Internet protocols suite, all the layers above the transport layer are grouped together. Someone could also say that only the application layer is used above the transport layer in practice. A famous application layer protocol is the *File Transfer Protocol (FTP)* that defines a standard way for a client and a server to transfer files between them. It should not be confused with the ftp software that is meant for end-users to transfer files and also implements the FTP protocol. Another typical application-specific protocol is the *HyperText Transfer Protocol (HTTP)* that is designed to manage and handle the transfer of Web pages remotely. It is implemented by Web browsers and Web servers, but also by systems that are not tied to the Web per se such as the Java's object-invocation mechanism.

2.2.2 Remote procedure call

A lot of distributed systems have been based on exchanging explicit messages between processes. However, this way of communicating does not conceal the communication at all, an important thing in order to achieve access transparency in distributed systems which was a well-known problem. A new way of handling communication was introduced around 1984 that allowed programs to call procedures located on other machines. When a process on machine A calls a procedure that exists on machine B, the execution of the called procedure happens on B, while the calling process on A is suspended. Any information needed can be transported from the caller to the callee in the form of parameters and can also come back as the procedure result. As a result, no message passing visible to the programmer at all, and this method is known as *Remote Procedure Call* or often just *RPC*. The basic idea sounds simple and elegant, but some subtle problems exist. Still most of them can be dealt with. RPC is a widely-used technique that underlies many distributed systems.

The main idea behind RPC is to make a remote procedure call seem like a local one. For example, a client calls a remote procedure like this: `result = remote_function(param1, param2)` which pretty much looks like a call to a function that exists in the local machine. A remote procedure call contains the following steps:

1. The client procedure calls the client stub (a different version of the remote procedure) using a normal calling sequence.
2. The client stub builds a message containing the parameters and calls the local operating system.
3. The client’s OS sends the message to the server’s OS.
4. The remote OS gives the message to the server stub (equivalent of client stub).
5. The server stub unpacks the parameters and calls the server procedure.
6. The server does the work needed and returns the result of the procedure call to the stub.
7. The server stub packs up the result in a message and calls the local OS.
8. The server’s OS sends the message back to the client’s OS.
9. The client’s OS delivers the message to the client stub.
10. The client’s stub unpacks the result of the remote procedure call and returns it to the client.

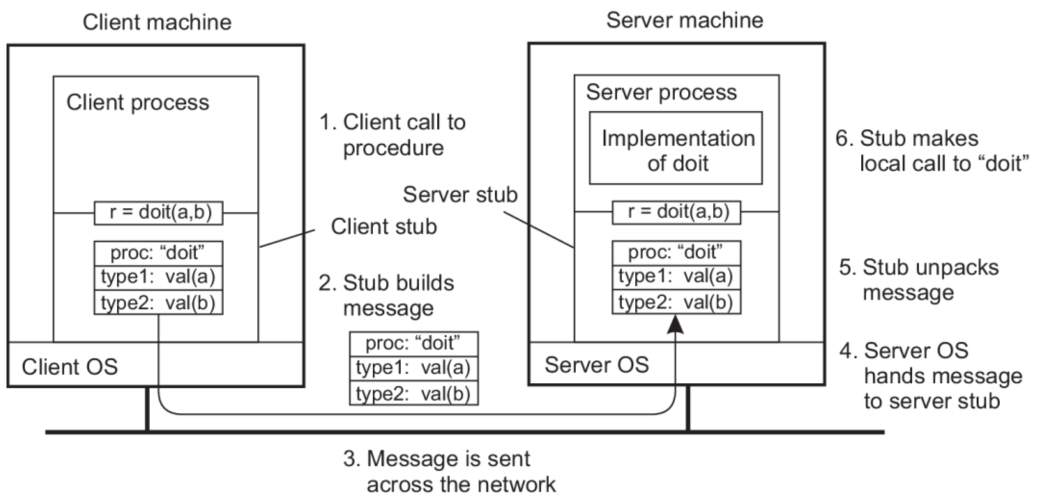


Figure 4. The steps involved in a remote procedure call `doIt(a, b)`. The return path is not shown

2.2.3 Message-oriented communication

While remote procedure calls contribute to hiding communication in distributed systems and enhance access transparency, this mechanism is not always appropriate. When the receiving side cannot be assumed to be executing at the time a request is sent, just like in RPC where the client is blocked until the request is processed, some alternative communication services are required. Messaging is the solution to this problem.

A lot of distributed systems are built directly on top of the simple message-oriented model offered by the transport layer, which is messaging through transport-level sockets. A socket is a communication end point to which an application can write data that are sent out over the underlying network, and from which incoming data can be read. A socket actually offers an abstraction over the actual port that is used by the local operating system for a specific transport protocol. The socket operations for TCP are:

- `socket`: The caller creates a new communication endpoint, meaning that the local OS reserves resources for sending and receiving messages.
- `bind`: Associates a local IP address to a newly created socket. The caller accepts messages only on the specified address and port.

- `listen`: Non-blocking call that allows the local OS to reserve resources for a specific maximum number of incoming connection requests to the caller.
- `accept`: Blocks the caller until a connection request arrives. The local OS creates a new socket with the same properties as the original one.
- `connect`: Attempts to establish a connection, provided the caller specifies the address to which the connection request should be sent.
- `send`: Sends data over the connection.
- `receive`: Receives data over the connection.
- `close`: Closes the connection.

Servers execute the first four operations in the order given, in general. Clients use the last four operations, with `close` also being called by both servers in order to fully terminate the connection.

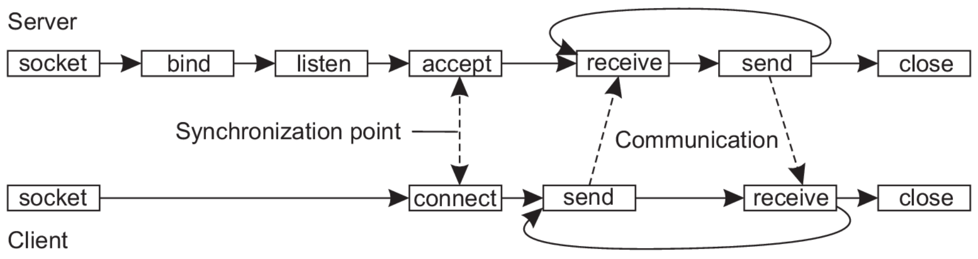


Figure 5. A connection-oriented communication using sockets

This standard approach with sockets is very basic and does not help making network programming easier or expanding beyond the currently offered functionalities. The development of *Message-Oriented-Middleware (MOM)* or generally known as *message-queuing systems* has helped a lot tackling those kind of problems. The basic idea of them is that applications communicate by inserting messages in specific queues. This happens by offering intermediate-term storage capacity for messages, without requiring either the sender or the receiver to be active when the message is transmitted; they can execute independently of each other. Each application has its own queue to which other applications can also send messages. That queue can only be read by the associated application. However, it can be possible for multiple applications to share a single queue. The basic interface that message queues offer to applications is the following:

- `put`: Called by a sender to append a message to the specified queue.
- `get`: Blocks until a specified queue is not empty, then removes the first message from the queue.
- `poll`: Removes the first message from the specified queue without blocking. If the queue is empty, the calling process continues.
- `notify`: Enables a handler to be called as a callback function when a message is put into the specified queue.

In order to integrate other applications into a single distributed system that handles communication via message queues, it is important that the different application can understand the different messages they receive. A sender and a receiver application must follow the same communication protocol in order to understand each other. Research has shown that this approach is not handled easily, so the best solution is to learn to live with those differences and try to make the conversions as easy as possible. These conversions are handled by *message brokers*, applications that convert incoming messages so that they are understood by the destination application.

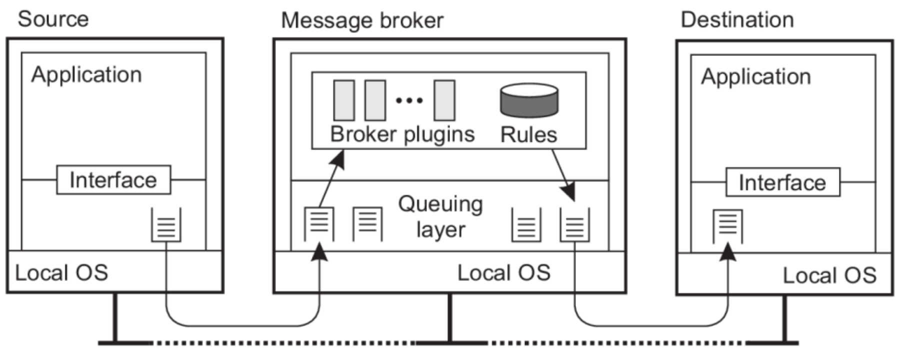


Figure 6. A message-queuing system with a message broker

Message queues that utilize a message broker follow the *publish-subscribe model* that will be discussed in the following paragraphs.

2.3 Architectural styles

Distributed systems are complex pieces of software of which the components are dispersed across multiple machines. To handle this type of complexity, it is essential that these systems are organized in a proper way. The organization of a distributed system's software components is called its *software architecture*. Components are directly related to connectors; mechanisms that are responsible for communication, coordination, cooperation, but also the flow of control and data between the individual components. The use of both components and their connectors can lead to various configurations that have been classified in specific *architectural styles*.

2.3.1 Layered architectures

In layered architectures components are arranged in a way that, a component at a higher layer can make a down-call at a component at a lower layer, usually expecting a response. In some cases, an up-call may have to be made to a higher layer from a lower layer.

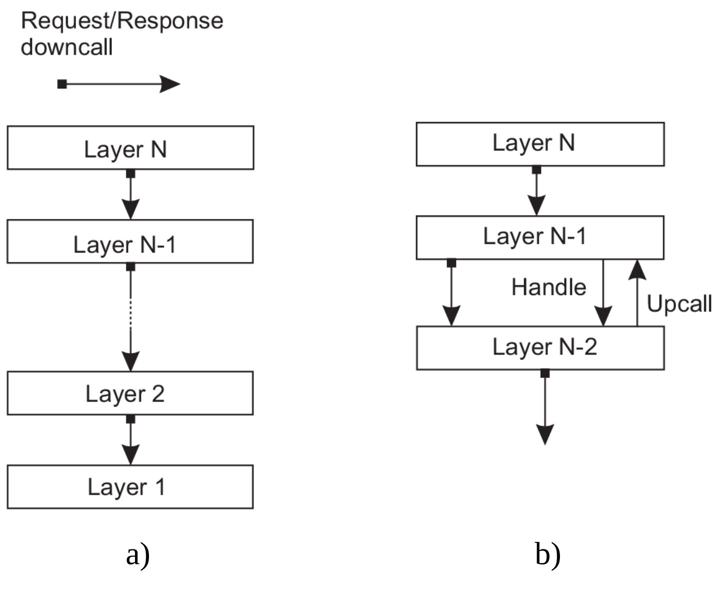


Figure 7. a) A pure layered organization b) A layered organizations with up-calls

The layered architectural style is very popular in the case of communication protocol stacks. The OSI model that was discussed in §2.2.1 is a pretty good example of the application of a layered architecture in the making of such communication infrastructure for distributed systems.

Regarding to the logical layering of applications, the most widely accepted layered organization comprises of three logical layers:

- *application-interface layer*: Takes care of the interaction with a user or another application.
- *processing layer*: The place where the core functionality of the application is implemented.
- *data layer*: Contains the database and the file system.

An example of such an application would be an Internet search engine. If the user interface is as simple as it gets, a user would type a few keywords inside a text box and by clicking a 'Search' button, a list of results with Web pages appears. The actual program that implements the search engine functionality is installed inside a server machine. It mutates the user submitted keywords into database queries, as well as creates a list of results that is rendered eventually in one or more HTML pages. Last, the database is filled with Web pages that have already been fetched and indexed.

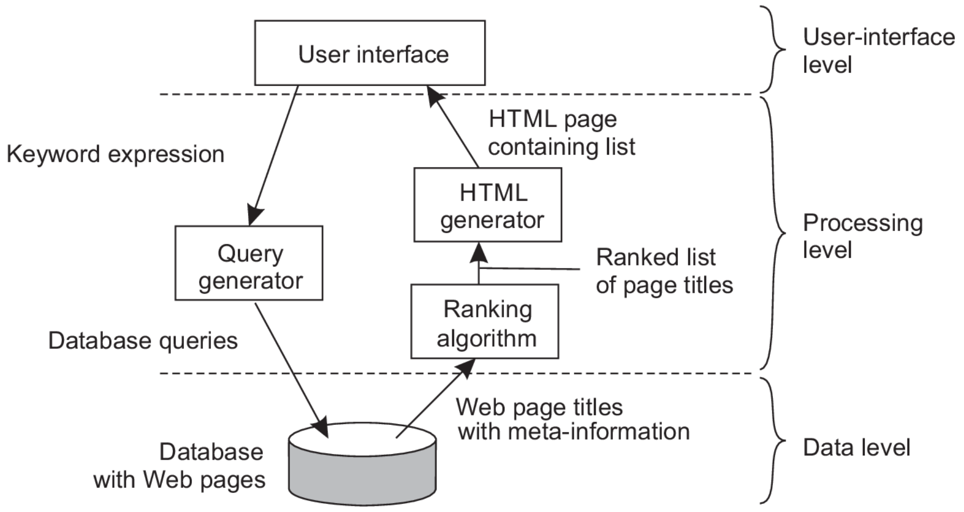


Figure 8. An Internet search engine organized in three different layers

2.3.2 Object-based architectures

A looser organization is implemented in object-based architectures, where components are in fact objects that are connected via procedure calls that take place over a network, so the calling object doesn't need to be on the same machine as the called object.

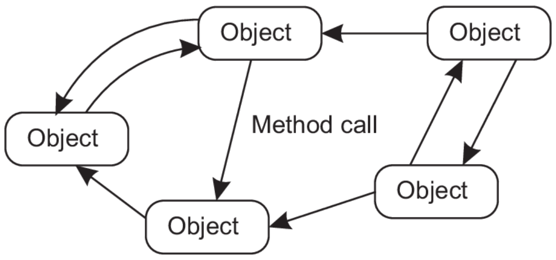


Figure 9. The object-based architecture

This type of architecture encapsulates data (an object's *state*) and functions to be called on the data (an object's *method*) into a single entity. Objects offer a communication *interface* that hides the implementation details of its functionalities. The interface alone can reside on one machine and the object itself with its state resides on a remote machine. In most distributed systems the state cannot be distributed. These characteristics are responsible for naming those objects *distributed* or *remote objects*.

A client machine can *bind* to a distributed object. When that happens, an implementation of the object's interface (analogous to client stub in RPC, generally called *proxy*) is loaded in the client machine's address space, marshaling method invocations into messages and unmarshaling reply messages from the server's object interface, which does the same marshaling-unmarshaling of messages as the client one's (analogous to server stub in RPC, generally called *skeleton*).

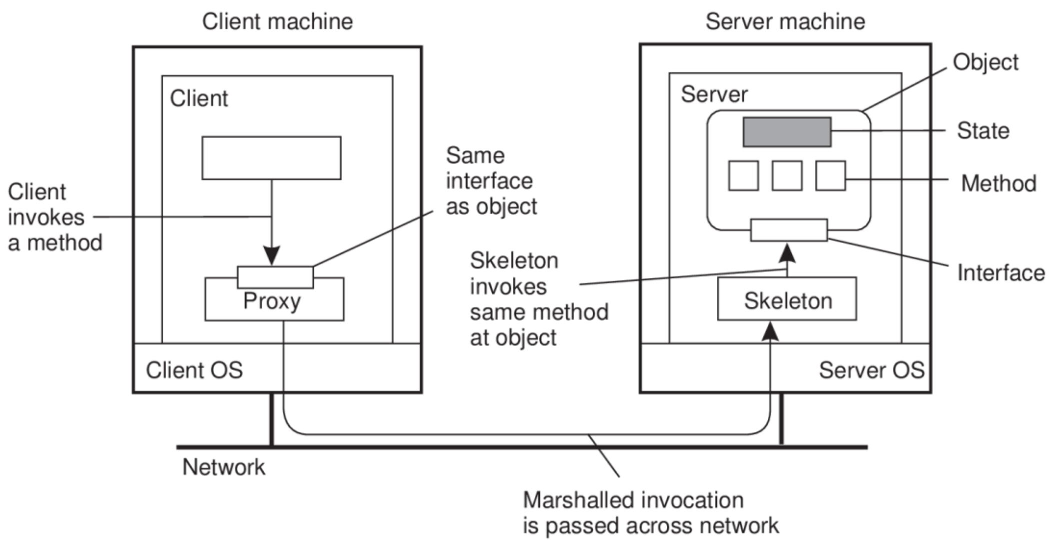


Figure 10. General organization of a remote object communication

An example of an actual remote object architecture is Java's take on RPC implementation, called *Remote Method Invocation* (*RMI*). In RMI a client in one virtual machine is able to invoke an object's method that exists inside another remote virtual machine.

2.3.3 Service-oriented architectures

Object-based architectures laid the foundation for encapsulating services into individual entities. These services are independent units that can operate on their own or make use of other services, a characteristic that brings the service-oriented architectures to the spotlight. A distributed application built with a service-oriented architecture consists of many different services, each having its own programming interface that allows it to communicate with other services with ease.

Most Web applications are built on top of service-oriented architecture; they are known by the term *web services*. According to the W3C Web Services Architecture Working Group, a web service "is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [84] [85].

Web services require the use of many layered and interrelated technologies. From a bottom-up perspective, the five layers of the web services architecture stack are the following [84] [85]:

- *the representation level*: It utilizes XML, which is the most popular language for representing data and messages sent between clients and web service providers [85].
- *the transport level*: Includes technologies used for data and messages transfer on the Web. HTTP is the most used one in data exchange between web Services. Also other technologies like TCP, SMTP and FTP are used as well [85].
- *the level of service invocation*: This level is about describing the structure of the data and messages sent to/from a web service. It actually specifies what XML elements are present inside a request to and a response from a Web service using the *Simple Object Access Protocol* (*SOAP*) [85].
- *the level of service description*: web services are described by specifying the interface through other services can communicate with it. This interface includes the available methods to call and its network location. Such interfaces are implemented in *Web Services Description Language* (*WSDL*) [85].
- *the level of service discovery*: Provides the means to a web service requester to find the desired web service based on certain criteria from services registries. This level takes advantage of the Universal Discovery Description Language (UDDI) [85].

2.3.4 Resource-based architectures

While the service-oriented architecture was the most popular style for distributed systems programming, at some point the number of the available services on the Web has increased dramatically, and so has the need for a new, more efficient way to connect and integrate services. An effective approach to this problem would be to conceive a distributed system as a big group of resources, each one being managed by a specific service where resources may be created, retrieved, updated or deleted (CRUD scheme).

This is known as *Representational State Transfer* (*REST*) and there are four characteristics that make a distributed system architecture a RESTful architecture:

- Every resource is reachable via a unique naming scheme named Uniform Resource Identifier (URI). The collection of all the URIs is called the Application Programming Interface (API) of the distributed system's software.
- All services provides the same interface and usually consists of four operations/methods of the HTTP protocol; POST for creating a new resource, GET for retrieving the state of the resource in a certain representation form, PUT for updating a resource by transferring a new state and DELETE for deleting a resource.
- The messages sent from or to a service are completely self-described.
- After a service is called and a specific function is executed, this service does not keep any information about the caller and can also change its state without being obliged to inform potential clients. This is also known as stateless execution.

In practice, if a client machine would like to call a RESTful web service maintained on a remote server, it would send an HTTP request with the POST, GET, PUT, DELETE methods in the unique service URI and wait for a response to be returned by it.

2.3.5 Publish-subscribe architectures

When a distributed system grows too much, so does the number of processes that join or leave it. I such cases, it is essential that the dependencies barrier between processes should lessen as much as possible, so that processes communicate more easily. To solve that problem, a lot of distributed systems have been designed so that they are in fact a collection of independent processes. With this architecture, there is a strong separation between *processing* and *coordination*, where the latter essentially glues together the actions of processes by handling their cooperation and communication.

Publish-subscribe architecture-based systems follow the referentially coupled-temporally coupled coordination model that leads to *event-based coordination*. Here processes are not aware of each other; they can only *publish* a notification that an event happened and also *subscribe* to a specific notification. Subscriber processes need to be alive the time the notification was published.

In those systems, communication happens by depicting events that a subscriber process is interested subscribing to. That means subscriptions must be coupled with certain type of notifications. As generally events are just data that become available, if a match between a publisher and subscriber happens, two situations may arise: either the middleware is going to forward the published notification with its data to the subscribed processes, or the middleware will forward a notification only at the point where subscribers are ready to receive the incoming data.

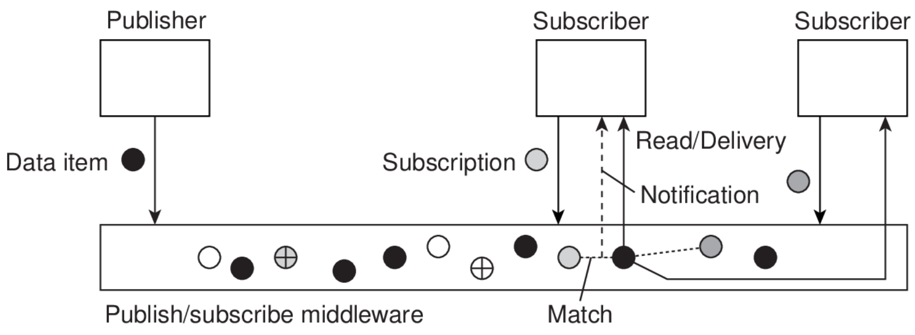


Figure 11. Data items exchanging between publishers and subscribers

A typical example of the publish-subscribe architecture is the implementation of the Message-Oriented Middleware (MOM), known mostly as message-queues, that was analyzed in §2.2.3.

2.4 Web applications and frameworks

The architectural styles mentioned above actually provide some basic middleware for developing applications to be taken advantage of by end-users [85]. Programmers can rely on this middleware for writing distributed systems software, of which the most popular is Web applications.

Although there is not an “official” definition, W3C describes a *web application* as “a Web page or collection of Web pages delivered over HTTP which use server-side or client-side processing to provide an application-like experience within a Web browser. Web applications are distinct from simple Web content in that they include locally executable elements of interactivity and persistent state” [86]. Web applications today are most important company assets utilized in business systems that are used in domains such as online e-banking, electronic marketplaces, stock market and e-trading and e-access to public administration services [87].

Web applications can be developed either by exploiting the existing low-level tools given by the selected architectural style/middleware, or by using a *web application framework* or just *web framework*. Server-side web frameworks are software frameworks that simplify writing, maintaining and scaling web applications [88]. They do so by abstracting away lower-level details of the interaction with the client [89], but also by providing tools and libraries that make easier carrying out common web development tasks, such as including routing URLs to appropriate handlers, interacting with databases, supporting sessions and user authorization, formatting output and offering security against web attacks [88].

Server-side web frameworks usually offer certain project structure conventions that affect the number and the content of its files and directories, as well as the communication between them. Two usual conventions are Model-View-Controller (MVC) and Model-Controller-Routes (MCR). In the MVC pattern, the Model part handles all the data and database-related operations of the application. The Controller part communicates with Model to retrieve the data needed and then processes them, thus containing the whole application logic. After that, the Controller produces the View, which handles the presentation of the data that will be shown to the client via a (graphical) user interface. The MCR pattern is mostly used for RESTful server-side applications that provide an API (e.g a collection of URIs in a RESTful architecture) that is to be consumed by a front-end or by another web service. The Model and Controller part hold the same roles as in the MVC pattern, but here the Controller actually defines some functions, where each one serves as a handler for a specific URI of the API.

Except for server-side web frameworks, client-side web frameworks also exist that have recently gained a lot of attention from the web programmers communities globally. These frameworks are concerned with programming browsers and enable programmers to write applications with rich and highly interactive user interfaces.

Programmers and organizations nowadays have the opportunity to develop web applications in the language of their choice, as server-side web frameworks exist for almost any programming language available. However, client-side web frameworks focus mostly on JavaScript, but there is an abundance of programming techniques and conventions that vary from framework to framework.

3 Comparison of Web Frameworks

3.1 Go for web applications

In the 1st chapter, many powerful aspects of the Go language have been analyzed. Many of these unique features can be used for the development of scalable, modular, maintainable, high-performance web applications.

Go provides a rich set of built-in standard libraries that are useful for various applications. A simple web application in Go can be written by just using the `net/http` and the `html/template` standard packages [90]. The Go language docs have very comprehensive information and numerous articles explaining how these packages work, but also how they can be used to create a simple web application.

Although these packages are very useful for web applications development, a modern web application may require some more advanced features that don't exist in the standard libraries. Furthermore, some people may find that the usage of these libraries require a fair amount of code to be written, so they would like to wrap the existing functionalities into simpler to use functions, whilst adding their own advanced features. This is how web application frameworks are born.

With Go having a large and continuously growing community, it was somewhat expected that some new web application frameworks would emerge. Right now there are reportedly more than 30 available open source web frameworks for Go [91].

3.2 Frameworks selection

This thesis' goal is to explore and compare the features that a selected number of Go web application frameworks provide. A total of five server-side frameworks where chosen and will be evaluated in the following paragraphs. But how where these five frameworks selected?

The main factor that was taken into account in the selection of the first three frameworks was their overall popularity. One way to measure a framework's popularity is to count how many GitHub stars its repository has. According to a GitHub repository that presents the most starred Go web frameworks on GitHub [91], the first three on the list are: `gin` with 38.5k stars, `beego` with 24k and `echo` with 17.3k stars. So `gin`, `beego` and `echo` are selected, since these are the most popular Go web frameworks right now. Two notable alternatives would be `kit` and `revel` that come behind `echo` with 17k and 11.7k stars respectively, web frameworks that are widely used and very popular in the Go community.

For the selection of the fourth framework, popularity and features completeness has been taken upon consideration. But this time the framework's popularity is selected to be significantly lower than the most popular ones. With 5.6k stars on GitHub, `buffalo` is behind the third most popular framework (`echo`). By browsing the `buffalo` documentation, it seemed that it was packed with a fair number of libraries and modules. The combination of the medium popularity along with the abundance of interesting features, led to the selection of the buffalo framework.

The last framework was found by exploring new and mainly unpopular Go web frameworks. `Iris` was created in 2016 and it is surprising the fact that although it is a framework that actually has a lot of GitHub stars, is rarely mentioned on the Web, but also when mentioned, it receives a lot of bad criticism. Eventually, `iris` is both popular and unpopular, so it would be really interesting to put it to the test. A possible alternative for `iris` might also have been the `macaron` framework [92].

3.3 Criteria for frameworks comparison

It is of high importance to determine the criteria which will be used for the evaluation of the selected frameworks.

The criteria can be divided into 2 main categories: Functional and Non-Functional requirements.

- *Functional requirement*: "defines a function or a software system or its component. A function is described as a set of inputs, the behavior and outputs" [93].
- *Non-Functional requirement*: refers to a concern not related to the functionality of the software [93].

The following table summarizes the functional and non-functional requirements that will be taken into consideration for the comparison of the web application frameworks. Note that those criteria were selected because they reflect some basic requirements of a modern web application. For a more complex application, some extra modules may be required for proper functionality, that either already exist in the current framework bundle, or they should be installed separately from the Administration-DevOps team or the individual developer.

Table 1. Functional and Non-Functional requirements

Functional	Non-Functional
Internationalization	*Convention over Configuration*
Testing	*Learning Curve*
Templating-Rendering	*Usability*
Validation	*Code Structure*
HTTP Client	*Security*
Caching	*Contribution*
Logging	*Community*
Static File Serving	*Extensibility*
File Upload-Download	*Performance*
Session-Cookies	
Authentication	
Database Handling	
E-mails	

The functional requirements are briefly described below:

- *Internationalization*: Module for providing the application in multiple languages.
- *Testing*: The framework has built-in support for building unit or API tests.
- *Templating-Rendering*: HTML Templating engine and ability to produce XML, JSON, YAML etc. file formats.
- *Validation*: Input, form or payload data validation.
- *HTTP Client*: An HTTP client is useful to make HTTP requests on demand.
- *Caching*: Module for caching data, making the application faster.

- *Logging*: Integrated logging module for documenting the server's and requests state.
- *Static File Serving*: Static files like HTML, JS, CSS, TXT, CSV etc. can be served.
- *File Upload-Download*: Support for uploading or downloading files to and from the server respectively.
- *Session-Cookies*: With a session, user-specific data can be stored in the server that are identified by a single cookie. Cookies themselves define and store user data in the client's browser.
- *Authentication*: For applications that have to do with accounts, user authentication is a crucial component.
- *Database Handling*: Built-in functions to make queries or transactions with a database, ORM support or other database related features.
- *E-mails*: Since a lot of websites communicate with their clients via e-mails, a mailing component is a good asset to the whole framework package.

A short description for every non-functional requirement is also given:

- *Convention over Configuration*: "All configuration should have sensible defaults, in order to minimize the number of detailed decisions that developers need to make for the common cases" [89]. Meaning that the framework should propose e.g. a fixed folder structure, a pattern like MVC and also make the developer avoid self-configuring the framework as much as possible.
- *Learning Curve*: How much time it will take to a developer who has never used the framework, to get familiar with it.
- *Usability*: The ease of use of the framework or how much does the framework help the developer use it effectively.
- *Code Structure*: The amount of lines of code it takes to write the application and the overall quality of the produced code after using the framework's features.

- *Security*: “It’s the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information” [93]. It usually has to do with the maturity of the framework, how long it has been used and tested by the developers in production environment or the amount of open issues in GitHub related to security flaws (if it’s an open source project). However, there could be some functional requirements related to security that the application might have, like SSL/TLS encryption, but overall, security can be more properly categorized as a non-functional requirement.
- *Contribution*: How often does the framework get updates from the publisher (if privately owned) or the community of developers (if open source and open for contribution).
- *Community*: Forum posts, articles and video tutorials on the Web, questions on StackOverflow and GitHub issues are factors that can define the overall developers’ interest for the framework.
- *Extensibility*: The ease of extending the framework’s API with new self-written features and modules.
- *Performance*: “It is an indication of the responsiveness of a system to execute any action within a given time interval” [93]. The performance can be measured by the amount of requests/sec, the total number of requests and also the average response time in a limited time frame. The average response time is the average total time needed for a server to process the request received and return the response to the client.

3.4 Comparison of frameworks

Having the comparison criteria defined, the next step is to see how the five selected frameworks meet them. But first these frameworks must be used and tested in a real case study.

The frameworks will be used to develop a CRUD web application with a RESTful API. That means, that the same application will be built with five different ways, each being dictated by each framework’s characteristics. Chapter 4 is dedicated to the development of the application where the idea and the development process is described in detail.

So, after the development of the application with each framework, the following results are obtained regarding the comparison criteria.

3.4.1 Internationalization

3.4.1.1 Gin

`Gin` doesn’t provide any internationalization modules yet. There is a new GitHub repository created for the development of a new internationalization middleware (`i18n`) in the main GitHub contribution page of `gin`, this being https://github.com/gin-contrib.

3.4.1.2 Beego

`Beego` provides an internationalization module that can be downloaded separately. The `github.com/beego/i18n` package can be used to provide a multilingual experience to the website users.

3.4.1.3 Echo

`Echo` does not provide any internationalization modules.

3.4.1.4 Buffalo

`Buffalo` uses the `github.com/nicksnyder/go-i18n` and `github.com/gobuffalo/mw-i18n` package for internationalization and localization of the application with a handful of useful functionalities. The translations are stored in the `locales` folder of the project.

3.4.1.5 Iris

Package `github.com/kataras/iris/i18n` enables internationalization and localization features for `iris`.

3.4.2 Testing

3.4.2.1 Gin

`Gin` encourages its users to take advantage of the `net/http/httptest` package of the standard library along with the `github.com/stretchr/testify` package as a dependency.

3.4.2.2 Beego

`Beego` makes it easy creating test cases using the `testing`, `net/http/httptest`, and some other helpful but not really necessary packages. When the `bee` CLI tool is used to create a `beego` project from scratch, a new `test` folder is created with a sample testing file. In addition, ORM tests for MySQL, SQLite3 and PostgreSQL databases can be made.

3.4.2.3 Echo

Test cases can be built using the `net/http/httptest` package of the standard library together with the `github.com/stretchr/testify`.

3.4.2.4 Buffalo

`Buffalo` has a dedicated test runner to make sure the test environment is correct and run the desired tests. It uses a mix of packages that enable convenient method, session and database setup testing. The tests can be run with the `buffalo test <package> -m <method>` command with the package argument being optional. It is also possible to generate test coverage reports.

3.4.2.5 Iris

`Iris` provides testing support with the `github.com/kataras/iris/httptest` package. This package provides helper functions to use `github.com/gavv/httpexpect` as an HTTP and REST API testing tool.

3.4.3 Templating-Rendering

3.4.3.1 Gin

`Gin` uses the `html/template` standard library package, but allows only one template to be rendered. To render multiple templates the `github.com/gin-contrib/multitemplate` package can be downloaded. Apart from HTML, `gin` also can render data in JSON, JSONP, ASCII-only JSON with escaped non-ASCII characters, XML and YAML format and also a plain String.

3.4.3.2 Beego

`Beego` uses the `html/template` standard library package. In startup the available templates are compiled and cached for efficient rendering. There is also the ability to pass data into the templates in a custom way by using `this.Data[]`, set template names, auto-render templates, change the default template tags, use built-in or custom template functions and control the whole layout design of the selected view. Apart from templates, `beego` can also render data in JSON, JSONP, XML and YAML format and also plain Strings from reading a `[]byte` array.

3.4.3.3 Echo

For template rendering, the `html/template` package is used, with the ability to call `echo` inside the templates. `Echo` can also render Strings, normal, beautified or stream JSON, JSONP, JSON Blob, normal, beautified or stream XML, XML Blob and also send files, attachments, Blob or just a data stream with the provided content type and the appropriate `io.Reader()`. Empty content rendering and special hook-type functions attachment are available as extra choices.

3.4.3.4 Buffalo

The `github.com/gobuffalo/buffalo/render package` has a collection of useful render options. They include auto format rendering, JSON, XML Markdown, JS and HTML rendering or just plain text. It's also possible to write custom renderers. For templating, the `github.com/gobuffalo/plush` package is used with features like standard layout setting, array, slice and map iterating, partials, helper functions and flash messages.

3.4.3.5 Iris

`Iris` supports 6 template engines, the standard `html/template`, the Django template parser, Pug, Handlebars, Amber and Jet with an extra ability to use any other template engine desired. Apart from templates, `iris` can also render data in JSON, JSONP, XML, YAML or Markdown format and also plain text.

3.4.4 Validation

3.4.4.1 Gin

`Gin` binds the request body into specific types to enable further processing. It currently supports binding of JSON, XML, YAML, and standard form values. It is also possible to build custom validators using the external `gokpg.in/go-playground/validator.v10` package.

3.4.4.2 Beego

`Beego` can check whether the input data are valid JSON, XML, YAML, standard form values, integers, booleans, floating point numbers and strings. The package `github.com/astaxie/beego/validation` can also be used for data validation and error collection.

3.4.4.3 Echo

`Echo` can check whether the payload data is valid JSON, standard form values and query parameters and bind them into a Go type, but it doesn't come with built-in data validation capabilities. The docs suggest using third-party libraries for data validation like `github.com/go-playground/validator`.

3.4.4.4 Buffalo

Package `github.com/gobuffalo/validate` enables data validation inside a buffalo application. `github.com/gobuffalo/validate/validators` has some predefined validators for basic data types validation. Also with the `github.com/gobuffalo/buffalo/binding` package certain types of payload data like JSON, XML, HTML or HTML forms can be bound to Go data types for easy handling.

3.4.4.5 Iris

For model binding and validation `Iris` uses the `github.com/go-playground/validator.v9` package. Validation can also happen in the url parameters. For example in the path `/users/{id:uint64} iris` checks if the provided `id` is an unsigned 64-bit integer, if not it throws a 404 error. Built-in functions but also user-defined ones can be used for more detailed validation of the url parameters.

3.4.5 HTTP Client

3.4.5.1 Gin

A custom http client is not provided with `gin`. Third-party ones can be used.

3.4.5.2 Beego

The `github.com/astaxie/beego/httplib` package is provided that enables the user to make HTTP or HTTPS calls, send big data, upload files but also to set headers, request parameters and timeout.

3.4.5.3 Echo

No http client package comes along with the `echo` framework bundle.

3.4.5.4 Buffalo

`Buffalo` doesn’t have a custom http client.

3.4.5.5 Iris

`Iris` does not provide a dedicated client to make HTTP calls.

3.4.6 Caching

3.4.6.1 Gin

The `gin` community contributed middleware `github.com/gin-contrib/cache` can be imported to use caching.

3.4.6.2 Beego

`Beego` supports caching via the `github.com/astaxie/beego/cache` package with 4 providers: memory, file, redis and memcache with the ability to create a custom provider by implementing the `Cache` interface.

3.4.6.3 Echo

Caching support is not provided. Third-party libraries or user-developed libraries can be used instead.

3.4.6.4 Buffalo

There is a caching plugin for `buffalo` but it is deprecated and not appropriate for use.

3.4.6.5 Iris

`Iris` supports both server-side and client-side caching. User sets a maximum data expiration time and after that time frame has passed, cached data are erased.

3.4.7 Logging

3.4.7.1 Gin

`Gin` provides a logger middleware that is highly customizable. There are also some community contributed logging packages as alternatives.

3.4.7.2 Beego

The `github.com/astaxie/beego/logs` is `beego`'s logging module. It can be imported and help with logging at multiple levels such as `LevelDebug`, `LevelNotice`, `LevelError`, `LevelCritical` and more.

3.4.7.3 Echo

`Echo` has a built-in logger middleware that can be set up with the `e.Use(middleware.Logger())` command and log information about every HTTP request. The logs can be configured to be shown in a specific format to the developer's liking.

3.4.7.4 Buffalo

`Buffalo` provides a logger that logs the requests made to the server and a general application logger. Also the Logger interface can be implemented to create a custom logger.

3.4.7.5 Iris

The `github.com/kataras/iris/middleware/logger` middleware enables HTTP request logging at multiple levels with configuration capabilities.

3.4.8 Static File Serving

3.4.8.1 Gin

There is support for defining paths that serve static file directories or even a single static file.

3.4.8.2 Beego

There is support for defining paths that serve static file directories or `even` a single static file. Multiple static files or directories can be registered together.

3.4.8.3 Echo

A new route can be registered to serve static files from a root directory with `e.Static()`. Also single static file serving is available via the `e.File()` command.

3.4.8.4 Buffalo

A new `render.Engine` with a new `packer.Box` can be defined to enable file serving.

3.4.8.5 Iris

Whole directories but also single static files can be served to the client with the appropriate function calls.

3.4.9 File Upload-Download

3.4.9.1 Gin

Single file uploading and downloading is available, with an extra option for multiple file uploading.

3.4.9.2 Beego

Single file downloading is available, as well as single or multiple file uploading.

3.4.9.3 Echo

One or more files can be uploaded with `echo`, but also there is support for downloading a single file.

3.4.9.4 Buffalo

A `render.Download()` renderer enables file downloading. Also files can be uploaded easily from a form.

3.4.9.5 Iris

`Iris` supports single/multiple file uploading and single file downloading.

3.4.10 Session-Cookies

3.4.10.1 Gin

`Gin` doesn't directly support sessions, but the `github.com/gin-contrib/sessions` package can be downloaded to use them. Instead, it pretty much supports setting and getting cookies.

3.4.10.2 Beego

`Beego` has a built-in session control module that supports memory, file, redis, MySQL, Couchbase, memcache and PostgreSQL. It also supports setting and getting simple cookies or cookies with encoded values.

3.4.10.3 Echo

For session management, the `echo` community maintained implementation can be taken advantage of, which is being backed by `github.com/gorilla/sessions`. Cookies are also available, but cookie-related functions functions require the use of the standard `http.Cookie`.

3.4.10.4 Buffalo

`Buffalo` sessions are available via the `buffalo.Context` and it uses `github.com/gorilla/sessions` package under the cover. Setting and getting cookies is also easy via the `Cookies()` method of the `buffalo.Context`.

3.4.10.5 Iris

Session storage and setting or getting cookies is available on `iris`.

3.4.11 Authentication

3.4.11.1 Gin

User authentication can be succeeded via the `gin.BasicAuth` middleware.

3.4.11.2 Beego

The package `github.com/astaxie/beego/plugins/auth` can provide basic user authentication. It is activated like a `beego` filter middleware with `beego.InsertFilter()`.

3.4.11.3 Echo

`Echo`'s basic auth middleware provides basic user authentication with a few configuration choices.

3.4.11.4 Buffalo

User authentication can be succeeded with the use of `github.com/gobuffalo/buffalo-auth` or the `github.com/gobuffalo/buffalo-goth` package. The second one provides authentication via social media accounts, which is a really good feature.

3.4.11.5 Iris

The `github.com/kataras/iris/middleware/basicauth` package can handle provide basic user authentication.

3.4.12 Database Handling

3.4.12.1 Gin

There aren't any database or data storage related tools inside `gin`.

3.4.12.2 Beego

`Beego` has a rich set of database related tools. The ORM tool supports drivers for MySQL, PostgreSQL and SQLite3, numerous operations like setting up a database, registering drivers, time zone configuration, setting maximum idle or open connections, generating tables, quick advanced filter-type queries and also the quick creation, retrieval, updating or deletion of an object without making any raw SQL queries, although it's possible to do so. Apart from ORM-related features, `beego` has a `QueryBuilder` asset that enables making fast, readable queries with ease and a. ORM tests can be made as mentioned in the Testing section.

3.4.12.3 Echo

Database-related features are absent from `echo`.

3.4.12.4 Buffalo

The `github.com/gobuffalo/pop` package is great for making ORM models, fast CRUD operations, run migrations, define associations and relationships between tables, execute database queries and define callbacks to be executed database operations. It supports MySQL, PostgreSQL, SQLite3 and CockroachDB and can be used conveniently along with the `soda` CLI tool.

3.4.12.5 Iris

`Iris` doesn't provide any database-related tools. However in the GitHub examples there is code that implements ORM logic with the `github.com/jinzhu/gorm` and `github.com/go-xorm/xorm` libraries.

3.4.13 Emails

3.4.13.1 Gin

`Gin` doesn't have a dedicated module for e-mail generation.

3.4.13.2 Beego

`Beego` has a dedicated module for e-mail generation and sending in `github.com/astaxie/beego/utils` package.

3.4.13.3 Echo

E-mail modules are not included in `echo`.

3.4.13.4 Buffalo

`Buffalo` provides a mailer extension that enables easy email generation with useful extra configurations.

3.4.13.5 Iris

There is no dedicated package for e-mail generation and sending.

3.4.14 Convention over Configuration

3.4.14.1 Gin

The framework doesn't propose a specific folder structure, on the other hand lacks any configuration-like philosophy. `Gin` has a default configuration setting and if the developer wants to use any extra middleware, he/she can include it in the current setting via the `Use()` method of the `gin` instance.

3.4.14.2 Beego

`Beego` uses a MVC architecture. Using the `bee` CLI tool, a new standard project or an API project can be created. Doing so, a specific folder structure is imposed on the whole project, that being divided in concrete logical entities. `Beego` also gives the developers the choice whether to implement a configuration logic to the application or not. The `app.conf` file contains all the custom configurations needed. An alternative way to do this would be to declare the desired configurations in the `main.go` file of the `beego` application.

3.4.14.3 Echo

`Echo` sets no boundaries in project folder structure and application architecture. It's a convention that if the developer needs to inject any desired middleware it can be done with the `Use()` method of the current `echo` instance. Overall, echo does not have a strong convention-like logic but it doesn't encourage configuration logic either.

3.4.14.4 Buffalo

When a new project is created with the `buffalo` CLI, a very specific and peculiar project structure is created, that has some similarities with the MVC pattern. The `actions` directory is the Controller, the `models` folder is the Model and the `templates` directory is the View. It contains some other directories as well like `assets`, `public`, grifts, locales, tmp and of course a main.go file. The buffalo framework has a strong configuration logic. There are some environment variable that can be set, with support for reading .env files for automatic project configuration. Even some functions, e.g. the creation of a new renderer or a new buffalo instance imposes a configuration procedure to be followed in order to set up the current application.

3.4.14.5 Iris

Iris does not propose any project structure, can be used to create applications with MVC architecture though. In general, iris pretty much works with a conventional way, from using middleware and dependency injection to defining and grouping routes. Configuration is only used for setting up the overall iris project from a built-in configurator, a configuration struct, a YAML or TOML file or some middleware like basicauth.

3.4.15 Learning Curve

3.4.15.1 Gin

The documentation is very good and guides the developer with examples how to use the framework’s features, leading to quick familiarization with it.

3.4.15.2 Beego

Extensive documentation with plenty of examples makes the learning process enjoyable and relatively quick. A downside would be that the official documentation doesn’t cover the whole spectrum of beego’s features, so if developers want to use some more advanced functionalities maybe they have to search on their own how to use them properly.

3.4.15.3 Echo

The documentation covers the biggest part of the echo features with understandable examples. Echo is really easy to learn and a basic application can be developed rapidly from the early stages of the learning process.

3.4.15.4 Buffalo

`Buffalo` docs have enough articles for the core features of the framework, these being a getting-started guide, examples for request handling, database manipulation, deployment practices but also various guides for advanced topics like setting background workers, building an API, plugins integration and use of `buffalo` with Docker images. Although the documentation is extensive, it requires some effort to learn the framework and get familiar to it.

3.4.15.5 Iris

The “Get Started” guide in `iris`’ official homepage (iris-go.com) gives a limited number of examples for basic usage of the framework. For more advanced usage, `iris` contains a `_examples` directory with actual code example of how to use the rest of `iris`’ features. It has a simple API which makes the learning process to take very little time. The process that someone would spend the most time on, would be searching the examples, just to see how the desired functionality should be implemented.

3.4.16 Usability

3.4.16.1 Gin

The provided functions greatly reduce the developer’s work. Especially their naming really helps with writing understandable code.

3.4.16.2 Beego

An abundance of useful functions reside in the `beego` framework. The MVC pattern along with the ease of use of the numerous tools make `beego` a really interesting framework to work with. Also the automated testing, packing and deploying are some of the strong points for usability purposes.

3.4.16.3 Echo

`Echo` has a really simple and straightforward API that boosts tremendously the overall usability of the framework. Without having a ton of features, it is a framework that makes a difference from an ease of use standpoint.

3.4.16.4 Buffalo

`Buffalo` does not give the usability that someone would expect from a Go web framework. It contains a handful of concepts and configuration practices that slow down the development process, despite having lots of good features for the application development.

3.4.16.5 Iris

Even for a beginner, `iris` is a very easy framework to use. The utilization of its features, even the customization of certain functionalities to the developers' needs is straightforward and facile.

3.4.17 Code Structure

3.4.17.1 Gin

In all 5 applications that were developed with the different frameworks, the file that contains database manipulation functions is the same and is 160 lines long with the break lines. If all lines of code (plus break lines) in all the files of the gin application are summed, there is a total of 366 lines of code. Having also in mind the benefits gained from the good usability of `gin`, the final product is elegant, concise and self-explanatory code.

3.4.17.2 Beego

If the break lines, the database manipulation file and all the other files except the tests are taken into account, there is a total of 548 lines of code. For someone to fully understand `beego` code, must be familiar with the naming conventions and concepts the framework uses. Having done that, the readability and expressiveness of the code is quite good.

3.4.17.3 Echo

Taking into account the constraints for lines of code counting that were mentioned previously, the application written in `echo` code is 358 lines long. The pros gained from its good usability directly affect the syntactic structure of the code as the final product is inclusive and elegant code.

3.4.17.4 Buffalo

Some directories of the standard project folder structure were deleted because they were of no use. Also there are some multiline comments that are useful for code readability. If no configuration files and directories with binary data are considered, the final code consists of 434 lines. If some effort is made, the final product can be quite clean and easily readable.

3.4.17.5 Iris

The code produced by using Iris is very comprehensive and thorough. The demo application developed consists eventually of 404 lines of code.

3.4.18 Security

3.4.18.1 Gin

`Gin`'s GitHub repository was created at 2014 and since then it has been used by a lot of people. It lacks any security issues, supports SSL and TLS, it has a stable API that is not going to change in future releases, can handle goroutine panics triggered by HTTP requests by recovering them and also can report such problems to `Sentry`, a real-time crash reporting software [94].

3.4.18.2 Beego

`Beego`'s is one of the first Go web frameworks that were created. As its development started at 2012, it has been used and tested in many commercial projects in production over the years. No security issues have been reported recently, it supports TLS and easy error handling. There is an extra option to enable live monitoring of the `beego` application as administrator.

3.4.18.3 Echo

`Echo`'s development started at 2015 and there are no open security issues in its GitHub page. In nearly 5 years of development, echo is now a mature framework that can be used without security concerns. TLS encryption is also an extra security feature.

3.4.18.4 Buffalo

`Buffalo`’s GitHub repository was created at 2014 and there were never any security issues reported. The `github.com/gobuffalo/mw-forcessl` middleware automatically forces a redirect to HTTPS instead of an HTTP. `Buffalo` also suggests that the developers place their application behind an Apache or Nginx proxy and let them do the “heavy SSL lifting”.

3.4.18.5 Iris

Since the `iris`’ GitHub repository creation in 2016, it has evolved significantly and continues to do so without having any known security issues so far. TLS encryption is available, so a server listening only to HTTPS can easily be started.

3.4.19 Contribution

3.4.19.1 Gin

The latest release v1.6.3 of `gin` was published on May 3, 2020. `Gin` contributors commit to the repository in a regular basis fixing existing bugs or adding new features.

3.4.19.2 Beego

Version v1.12.1 of the `beego` framework is the latest one available and has been released on February 7, 2020. Since this release, 8 commits have mediated with the latest commit having been published on June 1, 2020.

3.4.19.3 Echo

The latest version v4.1.16 of `echo` was published on March 30, 2020. Constant contribution to the project is a fact, with stable releases being released regularly.

3.4.19.4 Buffalo

The latest release v0.16.9 of `buffalo` was published on May 25, 2020 and commits take place about once a month.

3.4.19.5 Iris

Iris has had 56 releases published so far, with the latest one v12.1.8 having been released on February 16, 2020. The project’s contributors commit frequently, so new features are continuously added with great pace.

3.4.20 Community

3.4.20.1 Gin

With 2404 GitHub issues, 346 questions on StackOverflow [95] and numerous articles on the Web, `gin` has attracted a lot of attention from developers all around the world.

3.4.20.2 Beego

`Beego` has 4005 issues on its GitHub page, 498 questions on StackOverflow [96] and is referred frequently on articles on the Web. Those numbers indicate that there is a very active community interested in the `beego` project.

3.4.20.3 Echo

`Echo` has had 1584 issues on its GitHub page and 118 questions on StackOverflow [97]. Because of its simplicity, `echo` has urged a lot of developers to try it, so the Web is filling more and more with `echo` articles and video tutorials.

3.4.20.4 Buffalo

In total there have been published 2001 issues in GitHub, along with 74 questions on StackOverflow [98] and a fair number of articles and video tutorials on the Web.

3.4.20.5 Iris

With 1531 issues on GitHub, 72 questions on StackOverflow [99] and a growing number of references on articles and videos emerging are signs of an active community that grows day by day.

3.4.21 Extensibility

3.4.21.1 Gin

Anyone can develop their own middleware and integrate it easily with `gin` by utilizing the `Use()` method of the `gin.Engine`.

3.4.21.2 Beego

Custom middleware or even new independent packages can be developed and integrated into a `beego` project. For importing a custom built middleware the procedure described in `github.com/ojardila/beego_middleware` can be followed with the help of `beego.InsertFilter()`. A separate file for special handling functions can be created and then initialized in `routers.go` file.

3.4.21.3 Echo

Custom user middleware can be developed and integrated with the any `echo` application with the `Use()` method of the `echo` instance.

3.4.21.4 Buffalo

Developers can write their own middleware by implementing the `buffalo.MiddlewareFunc` interface. Buffalo plugin development is encouraged as well.

3.4.21.5 Iris

User-created handler functions for routes can be integrated into `iris` and used as custom middleware for the current application.

3.4.22 Performance

3.4.22.1 Gin

In the demo application that was developed, the requests that are actually used are the GET collection and POST entity, so the performance tests will be made only for these types of requests. It should be noted that in every GET request a total of 4655 database rows are fetched. The hardware used is an Intel i5-3210M @2.50GHz, 2 cores with each supporting 2 threads (4 logical cores), 4GB DDR3 RAM and a 500 GB SanDisk Ultra II SSD. Each framework's server runs on localhost in port 8080. After using the `github.com/tsliwowicz/go-wrk` package to test the framework, for approximately 5 seconds of benchmarking the results are the following.

- GET collection: 830 total requests, 165.84 requests/sec, 24.11987ms average request time.
- POST entity: 668 total requests, 133.64 requests/sec, 14.96603ms average request time.

3.4.22.2 Beego

After testing for 5 seconds with the `github.com/tsliwowicz/go-wrk` package, the results are the following.

- GET collection: 587 total requests, 117.22 requests/sec, 34.124682ms average request time.
- POST entity: 653 total requests, 130.62 requests/sec, 15.311725ms average request time.

3.4.22.3 Echo

The performance tests for 5 seconds come with the results below.

- GET collection: 896 total requests, 179.22 requests/sec, 22.318552ms average request time.
- POST entity: 831 total requests, 166.09 requests/sec, 12.041903ms average request time.

3.4.22.4 Buffalo

After running the performance tests, the following numbers are obtained.

- GET collection: 762 total requests, 152.08 requests/sec, 26.302248ms average request time.
- POST entity: 675 total requests, 134.98 requests/sec, 14.817061ms average request time.

3.4.22.5 Iris

Five seconds of benchmarking give the results shown below.

- GET collection: 773 total requests, 154.35 requests/sec, 25.914429ms average request time.
- POST entity: 655 total requests, 131.03 requests/sec, 15.263665ms average request time.

3.4.22.6 Performance Graphs

The benchmarks performed are summarized in the following table.

Table 2. Performance evaluation results for the 5 frameworks

	GET collection			**POST entity**		
	Total Requests	**Requests/sec**	**Avg. Request Time**	**Total Requests**	**Requests/sec**	**Avg. Request Time**
Gin	830	165.84	24.11987	668	133.64	14.96603
Beego	587	117.22	34.12468	653	130.62	15.31173
Echo	896	179.22	22.31855	831	166.09	12.0419
Buffalo	762	152.08	26.30225	675	134.98	14.81706
Iris	773	154.35	25.91443	655	131.03	15.26367

They can also be visualized for a better perception of the numbers.

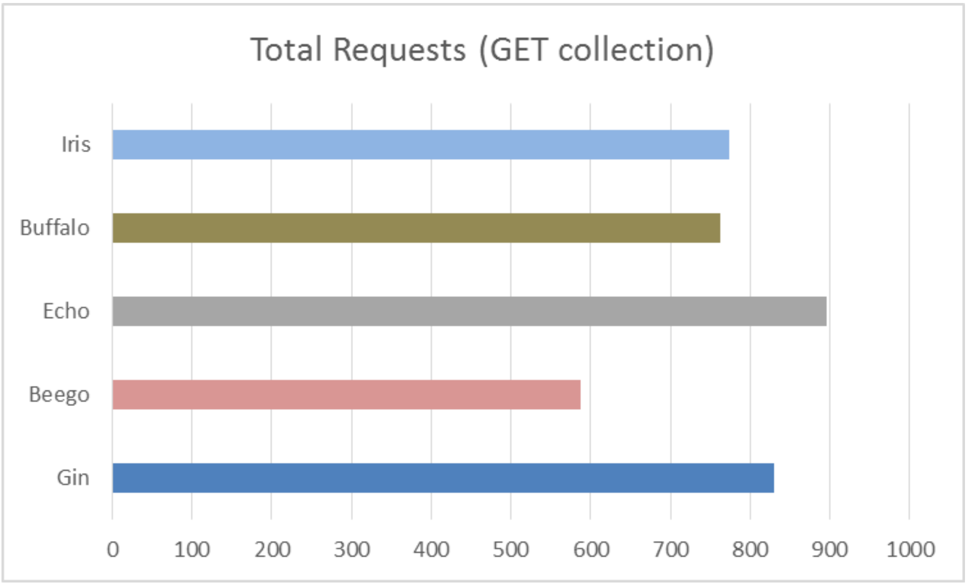


Figure 12. Total no. of Requests for GET collection

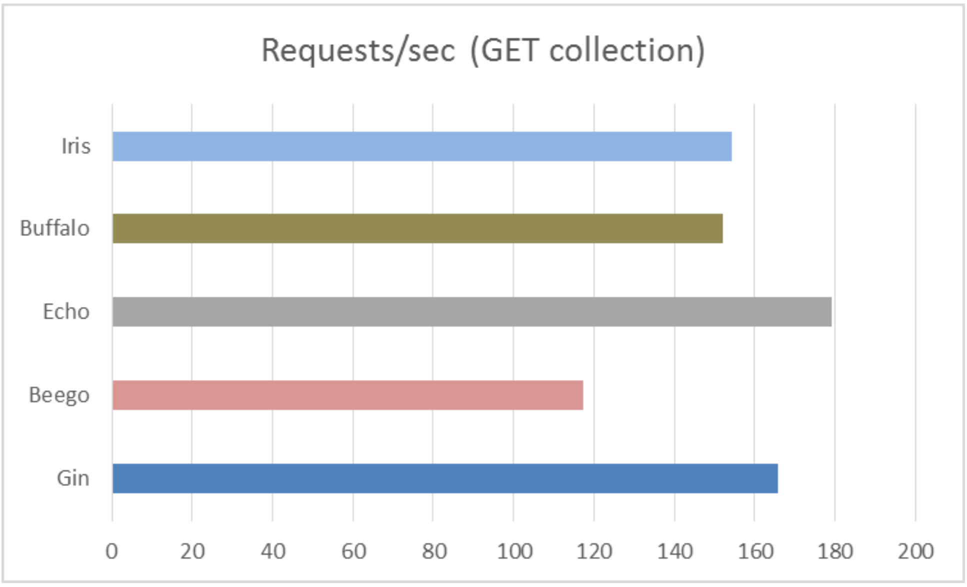


Figure 13. Requests/sec for GET collection

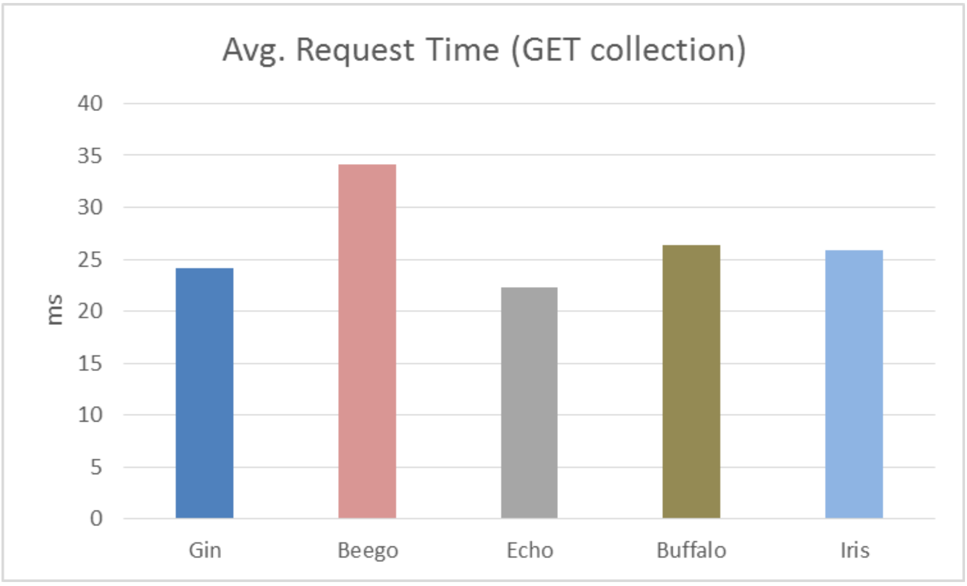


Figure 14. Average Request Time for GET collection

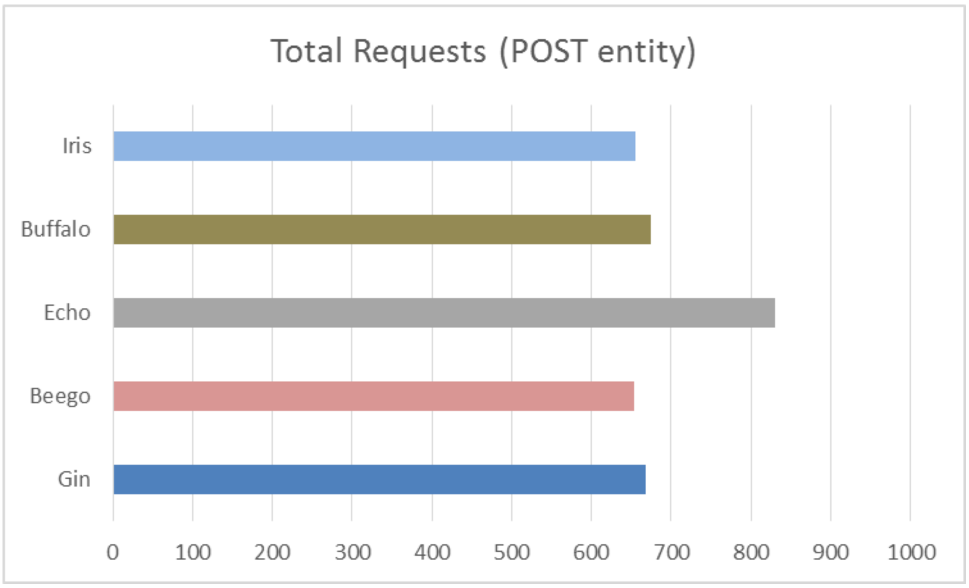


Figure 15. Total no. of Requests for POST entity

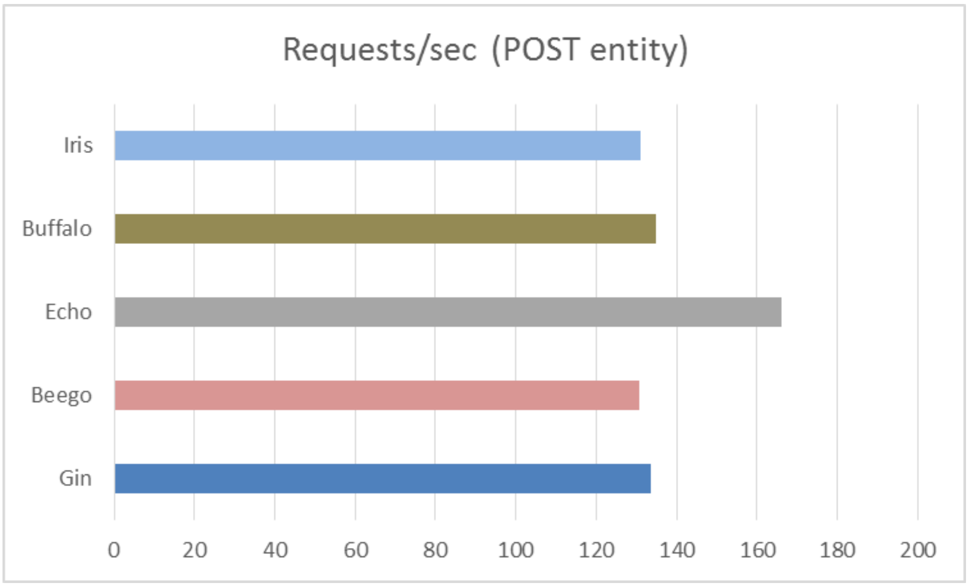


Figure 16. Requests/sec for POST entity

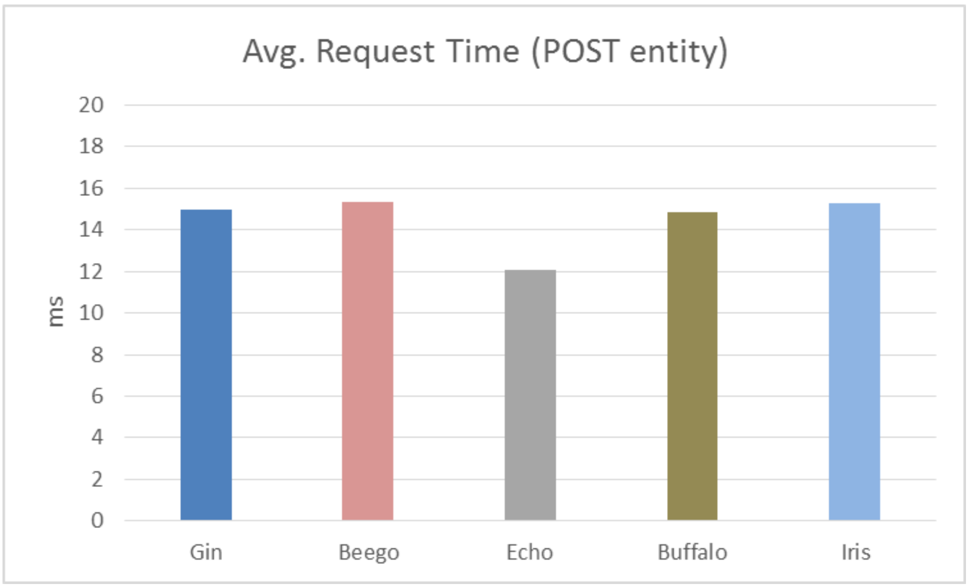


Figure 17. Average Request Time for POST entity

For GET collection requests, `echo` is the framework that dominates the other ones in performance aspect. `Gin` comes second and is being followed by `iris`, `buffalo` and `beego` in that order.

As regards to the POST entity requests, `echo` again is the framework that performs best. `Buffalo` is the second fastest framework leaving behind `gin`, `iris` and `beego` again in the order in which they were mentioned.

In GET requests it seems that the frameworks with the largest number of features score slower numbers than the lighter frameworks. But it in the POST requests, although the lightweight `echo` is first in terms of speed, `buffalo` comes second and `beego` last, so no safe conclusion can be extracted whether the more lightweight frameworks are actually faster than the heavier ones.

3.4.23 Summary

The overall comparison of the web frameworks is summarized in the table below:

Table 3. Overall comparison of web frameworks

	Gin	Beego	Echo	Buffalo	Iris
Internationalization	No	Optional	No	Optional – 3rd party	Yes
Testing	Standard library -3rd party	Yes – Standard library	Standard library -3rd party	Yes	Yes
Templating-Rendering	Yes -Standard library – Community pkg	Yes – Standard library	Yes – Standard library	Yes	Yes – Standard library
Validation	Yes – 3rd party	Yes	Yes – 3rd party	Yes	Yes – 3rd party
HTTP Client	No	Yes	No	No	No
Caching	Community pkg	Yes	No	Deprecated	Yes
Logging	Yes – Community pkg	Yes	Yes	Yes	Yes
Static File Serving	Yes	Yes	Yes	Yes	Yes
File Upload-Download	Yes	Yes	Yes	Yes	Yes
Session-Cookies	Yes – Community pkg	Yes	Community pkg	Yes	Yes
Authentication	Yes	Yes	Yes	Yes	Yes
Database Handling	No	Yes	No	Yes	3rd party
Emails	No	Yes	No	Yes	No
Convention over Configuration	Medium	High	Medium	Low	Medium
Learning Curve	Shallow	Rather steep	Shallow	Steep	Rather steep
Usability	Good	Medium	Good	Poor	Good
Code Structure	Good	Medium	Good	Medium	Good
Security	High	High	High	High	High
Contribution	Regular	Relatively regular	Regular	Sparse	Regular
Community	Large	Large	Large	Medium	Medium

Extensibility	Good	Good	Good	Good	Good
Performance	High	High	High	High	High

The green table entries that contain “3rd party” are about the 3rd party packages that each framework’s official documentation urges developers for use. If a framework does not recommend any 3rd party libraries, the appropriate table entry is marked red with “No”, even though there might be some pluggable packages.

The terms “Shallow” and “Steep” that describe the learning curve of a particular framework are used metaphorically, indicating that a framework with a “Shallow” learning curve requires less effort to learn than a framework with a “Steep” learning curve.

The fact that a framework has a lot of green entries in the table above does not necessarily mean that it is the best framework to choose overall. Something similar can also be said about frameworks with many orange or red entries too, these frameworks are not necessarily worse than others, as choosing the appropriate framework is a very subjective decision. For example, certain programmers are going to prefer an all-in-one framework like Beego with many features, while others may choose a more lightweight framework like Echo that does not have the same amount of features as Beego, but it is highly customizable and available for experimenting with third-party or self-written packages instead of having many built-in ones.

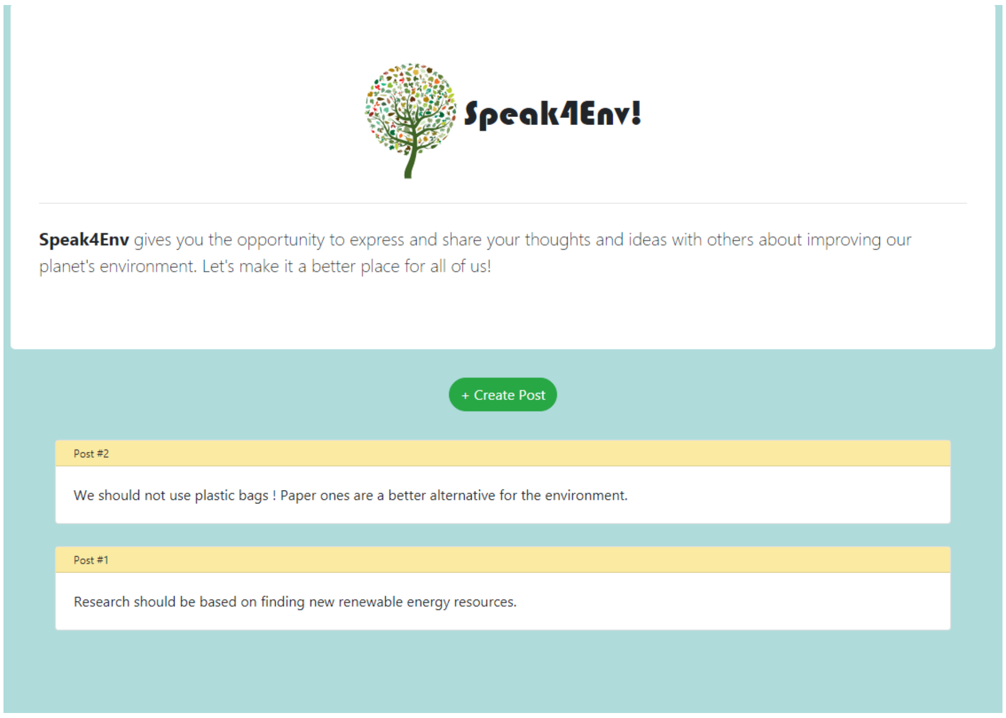
4 Demo Application

In Chapter 3, five Go web frameworks were selected to be compared according to specific criteria: `gin`, `beego`, `echo`, `buffalo` and `iris`. In order for them to be evaluated, it was considered necessary to use them for building an actual web application, as the only way to test every framework effectively, is to do it in practice.

4.1 The application's rationale

The main idea was to build a simple application that familiarizes the developer with the use of the five selected frameworks, with the ultimate goal of evaluating and comparing them. For that reason, it was decided that it would be a RESTful web application with basic CRUD operations that would also store data in a SQLite3 database.

The application's concept is the free online platform `Speak4Env`, which makes it easy for everyone to share their thoughts and ideas with other people around the world about improving the environment's quality.



Speak4Env gives you the opportunity to express and share your thoughts and ideas with others about improving our planet's environment. Let's make it a better place for all of us!

Post #2

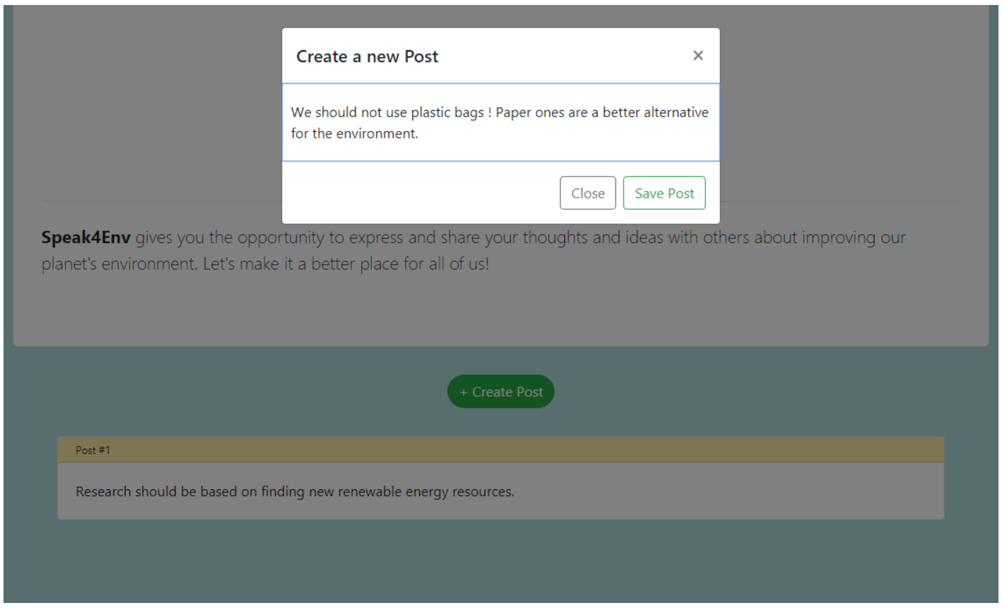
We should not use plastic bags ! Paper ones are a better alternative for the environment.

Post #1

Research should be based on finding new renewable energy resources.

Figure 18. The Speak4Env Application

In `Speak4Env`, the feature is that a user can anonymously publish a single Post at a time. By clicking the '+ Create Post' button, a modal dialog is shown where the Post's text can be typed.



We should not use plastic bags ! Paper ones are a better alternative for the environment.

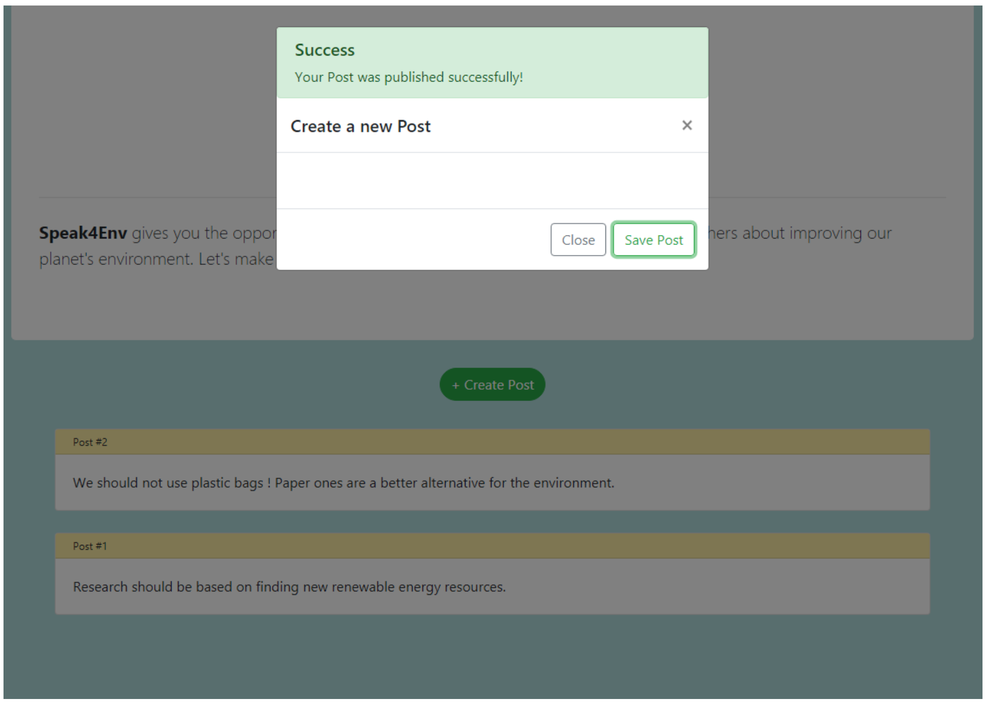
Speak4Env gives you the opportunity to express and share your thoughts and ideas with others about improving our planet's environment. Let's make it a better place for all of us!

Post #1

Research should be based on finding new renewable energy resources.

Figure 19. Creating a single Post

If the new Post is created successfully, a success message is shown to indicate it. Then the user can create another Post or just close the modal to read the rest of the published Posts.


Success
Your Post was published successfully!
Create a new Post
×
Close
Save Post
Speak4Env gives you the oppor
hers about improving our
planet's environment. Let's make
+ Create Post
Post #2
Post #1

We should not use plastic bags ! Paper ones are a better alternative for the environment.

Research should be based on finding new renewable energy resources.

Figure 20. Successful creation of a Post

When the main page is loaded, all the published Posts appear in the screen by chronological order, as shown in Figure 7.

It should be noted that, although the GET, PUT and DELETE entity web services were not implemented on the website, they were created in the backend in order to have a complete CRUD API.

4.2 Project structure

The project is based on both back-end and front-end code. The `backend` folder will be studied first and it contains 5 sub-folders, each for the application's code written with the help of the respective framework, along with a SQLite3 database file. The `frontend` folder will be studied later.

```
gofwc/
├── backend/
|   ├── beego/
|   ├── buffalo/
|   ├── echo/
|   ├── gin/
|   ├── iris/
|   └── maindb.db
└── frontend/
```

Figure 21. General project structure

In the following paragraphs the back-end directories will be analyzed in detail.

4.2.1 Gin

`Gin` is a framework that doesn't propose a specific project structure. For that reason, a Model-Controller-Routes structure convention is used for both logical and folder structure of the application.

```
gin/
├── controllers/
|   └── post.go
├── models/
|   └── datasource.go
├── routers/
|   └── router.go
├── README.md
└── main.go
```

Figure 22. Gin project structure

The `gin` folder contains the `controllers`, `models` and `routers` folders along with 2 separate files, `README.md` and `main.go`. The `models` folder has the datasource.go file that houses all the database related tasks that need to be done in the application. The post.go file inside the controllers folder is responsible for all the functions the back-end can handle, and also the router.go file of the routers folder makes those functions available via declaring the routes from where they can be accessed. README.md provides instructions on how to run the gin application and main.go is the starting point of the application where the gin instance is created, with the gin server listening to HTTP requests.

4.2.2 Beego

The beego API project is created automatically with the bee CLI tool, resulting in the following project structure:

```
beego/
├── conf/
|   └── app.conf
├── controllers/
|   └── post.go
├── models/
|   └── datasource.go
├── routers/
|   ├── commentsRouter…controllers.go
|   ├── commentsRouter_controllers.go
|   └── router.go
├── tests/
|   └── default_test.go
├── README.md
├── beego.exe
├── lastupdate.tmp
└── main.go
```

Figure 23. Beego project structure

Conf folder hosts the file responsible for the project configuration, while controllers, models and routers folders work the same way as in the gin project. The 2 extra files inside the routers folder are used for routing analysis when Annotation Router is used. The test folder contains test cases written to test if the application works properly. README.md file explains how to run the beego application, beego.exe is the executable created when bee run command is called to start the server, `lastupdate.tmp` is a cache file for the `Annotation Router` and `main.go` is used by the `bee run` command to enable the `beego` server listen to incoming requests.

The `bee` CLI tool also creates the `swagger` and `docs` folders for API documentation and an auto-generated document file respectively. They are deleted though because they are of no use to the application.

4.2.3 Echo

Like `gin`, `echo` doesn't have or propose a predefined structure for the application, so the Model-Controller-Routes convention is utilized.

```
echo/
├── controllers/
|   └── post.go
├── models/
|   └── datasource.go
├── routers/
|   └── router.go
├── README.md
└── main.go
```

Figure 24. Echo project structure

The folders' structure and content are identical to the `gin` application ones.

4.2.4 Buffalo

The `buffalo` API project is created with the framework's dedicated CLI tool and produces a unique folder structure.

```
buffalo/
├── actions/
|   ├── app.go
|   ├── handlers.go
|   └── render.go
├── config/
|   ├── buffalo-app.toml
|   └── buffalo-plugins.toml
├── models/
|   └── datasource.go
├── tmp/
```

```
|    └── buffalo-build.exe
├── .buffalo.dev.yml
├── .codeclimate.yml
├── .env
├── README.md
├── go
├── inflections.json
└── main.go
```

Figure 25. Buffalo project structure

`Actions` directory handles the Controller part of the Model-Controller-Routes pattern (`handlers.go` file), but also contains the `app.go` file for the application and routes setup and the `render.go` file to setup the desired `*render.Engine`. The `config` folder has the `buffalo-app.toml` file that documents the modules that have been installed when the project was created and the `buffalo-plugins.toml` file that does the same for all the `buffalo` plugins. `Models` folder is responsible for the Model part of the MCR pattern. `Tmp` folder is used by the `buffalo dev` command to rebuild the project on every change and all the temporary files that `buffalo` works with are put here [100]. The `.codeclimate.yml` and `.buffalo.dev.yml` are general configuration files, while the `.env` file handles environment variables declaration. The plain `go` file is automatically generated by the CLI tool. Finally, `README.md` and `main.go` files explain how to start the application and serve the `buffalo` application respectively.

Some other folders and files that were generated like `fixtures`, `grifts`, `Dockerfile`, `inflections.json` and `database.yml` were deleted.

4.2.5 Iris

`Iris` is a framework that provides MVC-oriented features. In the GitHub examples there are some MVC folder structures proposed for use. On the other hand, because the `iris` application is basically an API to be consumed, the previously mentioned Model-Controller-Routes structure will do just fine.

```
iris/
├── controllers/
|   └── post.go
├── models/
|   └── datasource.go
├── routers/
|   └── router.go
├── README.md
└── main.go
```

Figure 26. Iris project structure

4.2.6 Front-end

The front-end code was structured in such way that it can consume any of the 5 Go APIs that were built, meaning that only one frontend folder was created for the application.

```
frontend/
├── css/
|   └── speak4env.css
├── images/
|   ├── tree.ico
|   └── tree.svg
├── js/
|   └── speak4env.js
└── index.html
```

Figure 27. Front-end folder structure

The `css` folder contains the stylesheet that decorates the template of the application, `index.html`. The image shown on Speak4Env and the browser tab icon reside inside the `images` folder. The application needs a fair amount of JavaScript handling, so the `speak4env.js` file is created inside the `js` folder is created to do so.

4.3 Writing the application

All back-ends built with the selected Go frameworks are RESTful APIs, meaning that they provide some URIs available for consumption by a client who will call them, in order to perform a particular operation.

As explained in §4.1, these are going to be CRUD-like functions. They have to do with either a collection of items or single entities. So, the APIs are going to have the following 5 routes with their example JSON payloads and responses:

- 1 GET route to retrieve a collection of items

```
// Response
[
  {
    "ID": 2,
    "Text": "We should not use plastic bags ! Paper ones are a
better alternative for the environment."
  },
  {
    "ID": 1,
    "Text": "Research should be based on finding new renewable
energy resources."
  }
]
```

- 1 GET route to retrieve a single entity/item

```
// Response
{
  "ID": 1,
  "Text": "Research should be based on finding new renewable
energy resources."
}
```

- 1 POST route to create a single entity/item

```
// Payload
{
  "text": "Planting trees is a fun and useful way to protect the
environment."
}

// Response
{
  "postID": 3
}
```

- 1 PUT route to update a single entity/item

```
// Payload
{
  "text": "Updated text!"
}

// Response
{
  "post_updated": "yes"
}
```

- 1 DELETE route to delete a single entity/item

```
// Response
{
  "post_deleted": "yes"
}
```

After having decided what the RESTful routes should be, next step is to implement them in practice, along with the other components like the database manipulation and the whole web service logic.

4.3.1 Gin

As mentioned before, the `gin` project follows the MCR pattern. The Model file `datasource.go` handles all the data and database related functions that need to be done in the application.

`datasource.go`

```go
package models

import (
   "database/sql"
   "log"

   _ "github.com/mattn/go-sqlite3"
)

// The Post struct
type Post struct {
   ID   int
   Text string
}

var (
   db             *sql.DB
   driver         = "sqlite3"
   datasourceName =
"C:/Users/Evgenios/go/src/github.com/evgesoch/gofwc/backend/maindb.db"
)

// Open the database
func OpenDB() {…}

// Close the database
func CloseDB() {…}

// Create the database
func CreateDB() {…}

// Get all Posts from the database
func GetAllPosts() ([]*Post, error) {…}

// Get a Post from the database
func GetPostByID(id int) (*Post, error) {…}

// Create a new Post in the database
func CreatePost(text string) (int64, error) {…}

// Update a Post in the database
func UpdatePostByID(id int, text string) error {…}

// Delete a Post from the database
func DeletePostByID(id int) error {…}
```

First of all the `Post` type must be declared, because it will determine the database table's columns, but also the web services' payload and response format. Three other variables must also be declared, `db`, `driver` and `datasourceName` which are the database object, the database driver and the directory of the database file respectively. They will be put to use in the following functions:

- `OpenDB`: Opens a connection to the database.
- `CloseDB`: Closes the connection to the database.
- `CreateDB`: Used to create a database file and a table with specific columns.
- `GetAllPosts`: Fetches all published `Posts` from the database. Returns an array of `Posts` and an `error` value.
- `GetPostById`: Fetches a single `Post` from the database. Accepts the `Post`'s `id` as an argument and returns the fetched `Post` and an `error` value.
- `CreatePost`: Creates a new `Post` in the database. Accepts the `Post`'s `text` as an argument and returns the newly created `Post`'s id as an `int64` value, as well as an `error` value.
- `UpdatePostById`: Updates an existing `Post`'s text, given its `id` and the new `text`. Returns an `error` value.
- `DeletePostById`: Deletes a `Post` from the database. The `Post` for deletion is specified by its `id` and the function returns an `error` value.

The Model file is used inside the Controller file, `post.go`, which implements the whole logic of the application.

`post.go`

```go
package controllers

import (
   "log"
   "net/http"
   "strconv"

   ginModels "github.com/evgesoch/gofwc/backend/gin/models"
   "github.com/gin-gonic/gin"
)

// Get all Posts
func GetAllPosts() func(c *gin.Context) {…}
```

```go
// Get a Post
func GetPost() func(c *gin.Context) {…}

// Create a new Post
func CreatePost() func(c *gin.Context) {…}

// Update a Post
func UpdatePost() func(c *gin.Context) {…}

// Delete a Post
func DeletePost() func(c *gin.Context) {…}

// Check if a Post exists in the database by ID
func checkIfPostExists(postID int, postsSlice []*ginModels.Post) bool {…}
```

All the functions that appear inside the Controller (except one) return a `func(c *gin.Context)`, a handler function to be used in the Routes declaration and are analogous to the ones in the `database.go` file:

- `GetAllPosts`: Fetches all the published `Posts`.
- `GetPost`: Fetches a single `Post`.
- `CreatePost`: Creates a new `Post`.
- `UpdatePost`: Updates the text of an existing `Post`.
- `DeletePost`: Deletes a single `Post`.
- `checkIfPostExists`: A private-to-its-package helper function that performs a check if a specific `Post` exists. Used in `UpdatePost()` and `DeletePost()`, it accepts the `Post`'s id and a slice containing all the saved `Posts` and returns a `boolean` value.

The Routes file, `router.go`, uses the above functions to handle incoming requests to the API URIs available for consumption.

router.go

```go
package routers

import (
   ginControllers "github.com/evgesoch/gofwc/backend/gin/controllers"
   "github.com/gin-gonic/gin"
)

var r = gin.Default()

// Main Router for gin application
func SetupRouter() *gin.Engine {
   // Api
   r.GET("/posts", ginControllers.GetAllPosts())
   r.GET("/posts/:postID", ginControllers.GetPost())
   r.POST("/posts", ginControllers.CreatePost())
   r.PUT("/posts/:postID", ginControllers.UpdatePost())
   r.DELETE("/posts/:postID", ginControllers.DeletePost())

   // Frontend
   r.Static("/frontend", "../../frontend")
   r.StaticFile("/speak4env", "../../frontend/index.html")

   return r
}
```

Firstly a `*gin.Engine` instance is created. Then the basic routing function is declared, which contains the definition of both the API and the front-end routes. It returns the previous `*gin.Engine`.

The last part of the `gin` application is the `main.go`, that imports the Model and Router files, opens the database for read/write operations, initiates the routes and starts the `gin` web server which is listening to incoming client requests.

main.go

```
package main

import (
   ginModel "github.com/evgesoch/gofwc/backend/gin/models"
   ginRouter "github.com/evgesoch/gofwc/backend/gin/routers"
)

func main() {
   ginModel.OpenDB()

   r := ginRouter.SetupRouter()

   err := r.Run(":8080")
   if err != nil {
      ginModel.CloseDB()
      return
   }
}
```

4.3.2 Beego

The `beego` project may not have the same project structure as the `gin` one, but the main idea is the same again: the MCR pattern. The Model file is again `datasource.go` with the same functions and variables as in the `gin` application. The Controller file handling is different though, because of `beego`'s own internal structure.

post.go

```
package controllers

import (
   "encoding/json"
   "log"
   "strconv"

   "github.com/astaxie/beego"
   "github.com/evgesoch/gofwc/backend/beego/models"
)

// Main controller to hadle requests for Post
type PostController struct {
   beego.Controller
}

// @Title GetAll
// @Description Get all Posts
// @Success 200 {post} models.Post
// @router / [get]
func (pc *PostController) GetAll() {…}

// @Title Get
// @Description Get a Post by its ID
// @Param postID path int true "The Post's ID you want to get"
// @Success 200 {post} models.Post
// @router /:postID [get]
func (pc *PostController) Get() {…}

// @Title Post
// @Description Create a new Post
// @Param text body string true    "The new Post's text"
// @Success 200 {string} models.Post.Id
// @router / [post]
func (pc *PostController) Post() {…}

// @Title Put
// @Description Update a Post by its ID
// @Param postID path int true "The Post's ID you want to update"
// @Param text body string true    "The updated Post's text"
// @Success 201 {post} models.Post
// @router /:postID [put]
func (pc *PostController) Put() {…}

// @Title Delete
// @Description Delete a Post by its ID
// @Param postID path int true "The Post's ID you want to delete"
// @Success 204 {post} models.Post
// @router /:postID [delete]
func (pc *PostController) Delete() {…}

// Check if a Post exists in the database by ID
func checkIfPostExists(postID int, postsSlice []*models.Post) bool {…}
```

The `beego` Model file is imported because its functions will be used in the same manner as in the `gin` application. The first thing created is the main Controller, which is a `struct` that acquires all the methods of a `beego.Controller`. The next step is to assign the appropriate methods to the `PostController`, along with their annotation routing information. Annotation routing enables the developers to easily declare the data that will be associated with each Controller function, like success and failure messages, title and description of the interface, the route that is going to listen to requests, HTTP request method and also request parameters.

- `GetAll`: Fetches all the published `Posts`.
- `Get`: Fetches a single `Post`. Overrides the `beego.Controller Get()` method.
- `Post`: Creates a new `Post`. Overrides the `beego.Controller Post()` method.

- `Put`: Updates the text of an existing `Post`. Overrides the `beego.Controller Put()` method.
- `Delete`: Deletes a single `Post`. Overrides the `beego.Controller Delete()` method.
- `checkIfPostExists`: The helper function that performs a check if a `Post` with specific `id` exists.

Now that all the needed information is registered to the `PostController`, it is essential that a namespace router teams up with the `PostController` to create a new fixed route (namespace) on which the annotation routes will adjust. After that the front-end communication routes are also stated.

router.go

```go
package routers

import (
   "github.com/astaxie/beego"
   beegoController "github.com/evgesoch/gofwc/backend/beego/controllers"
)

func init() {
   // Api
   ns := beego.NewNamespace("/posts", beego.NSInclude(
      &beegoController.PostController{}),
   )
   beego.AddNamespace(ns)

   // Frontend
   beego.DelStaticPath("/static")
   beego.SetStaticPath("/frontend", "../../frontend")
   beego.SetStaticPath("/speak4env", "../../frontend/index.html")
}
```

The application is completed by writing the `main.go` file, which imports the Model and Router files, opens the database and initializes the `beego` server.

main.go

```go
package main

import (
   "os"
   "os/signal"
   "syscall"

   beegoModel "github.com/evgesoch/gofwc/backend/beego/models"
   _ "github.com/evgesoch/gofwc/backend/beego/routers"

   "github.com/astaxie/beego"
)

func main() {
   beegoModel.OpenDB()

   // Run the application
   if beego.BConfig.RunMode == "dev" {
      beego.BConfig.WebConfig.DirectoryIndex = true
      beego.BConfig.WebConfig.StaticDir["/swagger"] = "swagger"
   }
   beego.Run()

   handleServerTermination()
}

// Close the database after server termination
func handleServerTermination() {…}
```

4.3.3 Echo

The echo application has exactly the same structure as the gin one. Following the MCR pattern, again the Model file `datasource.go` is the same. The Controller file is the following:

`post.go`

```
package controllers

import (
   "log"
   "net/http"
   "strconv"

   echoModels "github.com/evgesoch/gofwc/backend/echo/models"
   "github.com/labstack/echo"
)

// Get all Posts
func GetAllPosts() func(c echo.Context) error {…}

// Get a Post
func GetPost() func(c echo.Context) error {…}

// Create a Post
func CreatePost() func(c echo.Context) error {…}

// Update a Post
func UpdatePost() func(c echo.Context) error {…}

// Delete a Post
func DeletePost() func(c echo.Context) error {…}

// Check if a Post exists in the database by ID
func checkIfPostExists(postID int, postsSlice []*echoModels.Post) bool {…}
```

The functions included in the Controller file have the same name as the ones in the gin application, meaning we have one function for fetching all the `Posts`, one for fetching a single `Post`, one for creating a new `Post`, one for updating a `Post`’s text and one for deleting a `Post`. These functions return each a `func(c echo.Context) error`, which is a function that returns an error value. They are going to be used in the Routes file as handler functions for the appropriate routes.

`router.go`

```go
package routers

import (
   echoControllers "github.com/evgesoch/gofwc/backend/echo/controllers"
   "github.com/labstack/echo"
)

// Setup the routes for the echo application
func SetupRoutes(e *echo.Echo) {
   // Api
   e.GET("posts", echoControllers.GetAllPosts())
   e.GET("posts/:postID", echoControllers.GetPost())
   e.POST("posts", echoControllers.CreatePost())
   e.PUT("posts/:postID", echoControllers.UpdatePost())
   e.DELETE("posts/:postID", echoControllers.DeletePost())

   // Frontend
   e.Static("/frontend", "../../frontend")
   e.File("/speak4env", "../../frontend/index.html")
}
```

The final stage is the creation of the `main.go` file, where the database connection opens, a new `echo` instance is created, the declared Routes are assigned to the `echo` instance and the web server starts listening to HTTP requests.

`main.go`

```go
package main

import (
   echoModel "github.com/evgesoch/gofwc/backend/echo/models"
   echoRouter "github.com/evgesoch/gofwc/backend/echo/routers"

   "github.com/labstack/echo"
)

func main() {
   echoModel.OpenDB()

   e := echo.New()

   echoRouter.SetupRoutes(e)

   err := e.Start(":8080")
   if err != nil {
      echoModel.CloseDB()
   }

   e.Logger.Fatal(err)
}
```

4.3.4 Buffalo

The `buffalo` framework adopts a very unique project structure. However there are some notable similarities with the other frameworks' structures. Starting from the Model part, the existing `datasource.go` file can be used as whole in this case. Next step is to define the default fall-back `Content-Type` header by setting the default `buffalo` render engine in `render.go` file.

render.go

```
package actions

import (
   "github.com/gobuffalo/buffalo/render"
)

var r *render.Engine

func init() {
   r = render.New(render.Options{
      DefaultContentType: "application/json",
   })
}
```

The equivalent Controller part of the `buffalo` application is the `handlers.go` file, where the Routes' handler functions are created.

handlers.go

```
package actions

import (
   "log"
   "strconv"

   buffaloModels "github.com/evgesoch/gofwc/backend/buffalo/models"
   "github.com/gobuffalo/buffalo"
   "github.com/gobuffalo/buffalo/render"
   packer "github.com/gobuffalo/packr"
)

// Get All Posts
func GetAllPosts(c buffalo.Context) error {…}

// Get a Post
func GetPost(c buffalo.Context) error {…}

// Create a Post
func CreatePost(c buffalo.Context) error {…}

// Update a Post
func UpdatePost(c buffalo.Context) error {…}

// Delete a Post
func DeletePost(c buffalo.Context) error {…}

// Serve the index page
func GetIndexPage(c buffalo.Context) error {…}

// Check if a Post exists in the database by ID
func checkIfPostExists(postID int, postsSlice []*buffaloModels.Post) bool {…}
```

These handler functions are similar to those of the `gin` and `echo` Controllers and return an `error` value. In the `buffalo` case, it is deemed necessary that a separate handler function `GetIndexPage` is to be created for the front-end routes handling because a new Renderer needs to be created in order to serve the webpage's HTML file.

Next step is to fill in the `app.go` file with the application's routes and the middleware used.

app.go

```
package actions

import (
   "net/http"

   "github.com/gobuffalo/buffalo"
   "github.com/gobuffalo/envy"

   contenttype "github.com/gobuffalo/mw-contenttype"
   "github.com/gobuffalo/x/sessions"
   "github.com/rs/cors"
)

// ENV is used to help switch settings based on where the
// application is being run. Default is "development".
var ENV = envy.Get("GO_ENV", "development")
var app *buffalo.App


// …
func App() *buffalo.App {
   if app == nil {
      app = buffalo.New(buffalo.Options{
         Env:          ENV,
         SessionStore: sessions.Null{},
         PreWares: []buffalo.PreWare{
            cors.Default().Handler,
         },
         SessionName: "_buffalo_session",
      })

      // Set the request content type to JSON
      app.Use(contenttype.Set("application/json"))

      // Api
      app.GET("/posts", GetAllPosts)
      app.GET("/posts/{postID}", GetPost)
      app.POST("/posts", CreatePost)
      app.PUT("/posts/{postID}", UpdatePost)
      app.DELETE("/posts/{postID}", DeletePost)

      // Frontend
      app.ServeFiles("/frontend", http.Dir("../../frontend"))
      app.GET("/speak4env", GetIndexPage)
   }

   return app
}
```

The `App()` function that creates a new `*buffalo.App` is called inside `main.go` file where the database opens and the buffalo server is started.

main.go

```
package main

import (
   "log"

   buffaloModel "github.com/evgesoch/gofwc/backend/buffalo/models"

   "github.com/evgesoch/gofwc/backend/buffalo/actions"
)

// …
func main() {
   buffaloModel.OpenDB()
   app := actions.App()
   if err := app.Serve(); err != nil {
      buffaloModel.CloseDB()
      log.Fatal(err)
   }
}

/* … */
```

4.3.5 Iris

The `iris` project has identical structure to the `gin` and `echo` projects. The `datasource.go` file is the same and handles the database operations, while the `post.go` file implements the web services' main functionality.

post.go

```
package controllers

import (
   "log"
   "strconv"

   irisModels "github.com/evgesoch/gofwc/backend/iris/models"
   "github.com/kataras/iris"
)

// Get All Posts
func GetAllPosts() func(ctx iris.Context) {…}

// Get a Post
func GetPost() func(ctx iris.Context) {…}

// Create a Post
func CreatePost() func(ctx iris.Context) {…}

// Update a Post
func UpdatePost() func(ctx iris.Context) {…}

// Delete a Post
func DeletePost() func(ctx iris.Context) {…}

// Serve the index page
func GetIndexPage() func(ctx iris.Context) {…}

// Check if a Post exists in the database by ID
func checkIfPostExists(postID int, postsSlice []*irisModels.Post) bool {…}
```

The handler functions found inside are again the ones that incarnate the CRUD character of the application. The extra function `GetIndexPage` serves as a handler for the route that provides the HTML template from server-side.

The `iris` Routes are declared inside the `SetupRoutes` function that takes a `*iris.Application` instance as an argument.

router.go

```
package routers

import (
   irisControllers "github.com/evgesoch/gofwc/backend/iris/controllers"
   "github.com/kataras/iris"
)

func SetupRoutes(app *iris.Application) {
   // Api
   app.Get("/posts", irisControllers.GetAllPosts())
   app.Get("/posts/{postID}", irisControllers.GetPost())
   app.Post("/posts", irisControllers.CreatePost())
   app.Put("/posts/{postID}", irisControllers.UpdatePost())
   app.Delete("/posts/{postID}", irisControllers.DeletePost())

   // Frontend
   app.HandleDir("/frontend", "../../frontend")
   app.Get("/speak4env", irisControllers.GetIndexPage())
}
```

The final piece of the `iris` application is the `main.go` file.

main.go

```
package main

import (
   irisModel "github.com/evgesoch/gofwc/backend/iris/models"
   irisRouter "github.com/evgesoch/gofwc/backend/iris/routers"

   "github.com/kataras/iris"
   "github.com/kataras/iris/middleware/logger"
   "github.com/kataras/iris/middleware/recover"
)

func main() {
   irisModel.OpenDB()

   app := iris.New()
   app.Logger().SetLevel("debug")
   // …
   app.Use(recover.New())
   app.Use(logger.New())

   irisRouter.SetupRoutes(app)

   err := app.Run(iris.Addr(":8080"),
iris.WithoutServerError(iris.ErrServerClosed))
   if err != nil {
      irisModel.CloseDB()
   }
}
```

4.3.6 Front-end

The front-end consists of HTML, CSS, JS and image files. The `Speak4Env` application can be used no matter what web server is listening (e.g. `gin` server, `echo` server), because each one is configured to just serve the static front-end files.

The HTML template `index.html` imports the CSS stylesheet `speak4env.css` that contains all the styling rules of the HTML elements and the 2 images, `tree.ico` and `tree.svg`.

speak4env.css

```
.body {…}
.treeLogo {…}
.s4e-title {…}
.postText {…}
.s4e-buttonContainer {…}
.s4e-buttonContainer button {…}
.s4e-postHeader {…}
.postsContainer {…}
#backToTop {…}
.errorAllPostsMessage {…}
.s4e-spinner {…}
```

It also imports the JS file `speak4env.js` that manages the communication with the server, but also the animations and HTML elements handling when a user interacts with the application.

`speak4env.js`

```
fetchAndRenderAllPosts();

$(function() {
   listenAndActOnSavePostButtonPress();
   scrollPageToTop();
});

/**
 * Fetch all Posts from the database and display them
 */
function fetchAndRenderAllPosts() {…}

/**
 * Respond to clicks on Save Post button in the modal and create a new Post
 */
function listenAndActOnSavePostButtonPress() {…}

/**
 * Prepare data for creating a new Post
 *
 * @param {String} postText The new Post's text
 *
 * @return {Object}         JS object with payload data
 */
function prepareNewPostData(postText) {…}

/**
 * Wrapper for jQuery $.ajax
 *
 * @param {String} method The HTTP method
 * @param {String} url    The url in which the request is sent
 * @param {JSON}   data   The payload data
 *
 * @return {jqXHR}        jQuery XHR object
 */
function makeAjaxRequest(method, url, data) {…}

/**
 * Refresh the Posts every 5 sec
 */
```

```
function fetchAllPostsEvery5Sec() {…}

/**
 * Scroll the page to top
 */
function scrollPageToTop() {…}
```

Inside the file there the following functions:

- `fetchAndRenderAllPosts`: Initially an AJAX call is made to the GetAllPosts web service to fetch the data regarding all the Posts in the database when the application is loaded for the first time. If the AJAX request is successful, the Post card elements appear one on top of another on the application window.
- `listenAndActOnSavePostButtonPress`: An `onclick` event listener for the "Save Post" button. When clicked, an AJAX request is made to the CreatePost web service and saves the newly created Post. If the request is successful, the new Post is rendered on top of the other ones.
- `scrollPageToTop`: There is a case the number of Posts grows to be big enough to make it difficult to scroll to the top of the page. This function is listening for both an `onscroll` event that gets triggered when the user scrolled after a certain position in the page to show the "Top" button and an `onclick` event assigned to that button that when triggered, automatically scrolls the page to the top.
- `prepareNewPostData`: It prepares the payload for the AJAX call to the CreatePost web service.
- `makeAjaxRequest`: This is a wrapper function that simplifies the call to jQuery's `$.ajax` function.
- `fetchAllPostsEvery5Sec`: It calls `fetchAndRenderAllPosts` with an interval of 5 seconds for automatic refresh of the Posts list. Currently not used in the application.

The jQuery and Bootstrap 4 libraries are used via CDN for more comfortable JavaScript handling, but also some predefined styling and behavior for specific elements of the site, respectively. The cards and modal are such elements.

The index.html is presented below:

index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
   <!-- Dependencies without cdn -->
   <!-- … -->

   <!-- Dependencies with cdn -->
   <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
      integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
   <script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
      integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
      crossorigin="anonymous"></script>
   <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
      integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
      crossorigin="anonymous"></script>
   <link rel="stylesheet" href="frontend/css/speak4env.css">
   <meta charset="utf-8" />
   <link rel="icon" href="frontend/images/tree.ico" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
   <meta name="description" content="Speak4Env lets you express your ideas about improving the
environment." />
   <title>Speak4Env!</title>
   <!--Tree image source: https://pixabay.com/vectors/abstract-art-colorful-ecology-leaf-1751248/-->
</head>

<body class="body">
   <noscript>You need to enable JavaScript to run this app.</noscript>

   <!-- Header -->
   <div id="header" class="container">
      <div class="jumbotron bg-white mb-10">
         <div class="container text-center">
            <div class="d-inline text-center">
               <img src="frontend/images/tree.svg" class="treeLogo" alt="Speak4Env logo"></img>
            </div>
            <div class="d-inline s4e-title h1">
               Speak4Env!
            </div>
         </div>
         <hr class="my-4" />
         <p class="lead"><span class="font-weight-bold">Speak4Env</span> gives you the opportunity to
express and
            share your thoughts and ideas with others about improving our planet's environment. Let's
make it a
            better place for all of us!</p>
      </div>
   </div>

   <!-- Posts cannot load error alert-->
   <div class="container errorAllPostsMessage">
      <div class="alert alert-danger d-none mb-0" id="errorAllPostsMessage" role="alert">
         <h5 class="alert-heading">
            Error
         </h5>
         <p class="mb-0">
            An error occurred while fetching the Posts. Please refresh the page.
         </p>
      </div>
   </div>

   <!-- Create Post Button-->
   <div class="container s4e-buttonContainer text-center">
      <button id="create-post-btn" type="button" class="btn btn-success rounded-pill" data-
toggle="modal" data-target="#createPostModal" disabled="true">
         + Create Post
      </button>
   </div>

   <!-- Loading spinner -->
   <div id="spinner" class="text-center">
      <div class="spinner-grow text-secondary s4e-spinner" role="status">
         <span class="sr-only">Loading...</span>
      </div>
   </div>

   <!-- Modal -->
   <div class="modal fade" id="createPostModal" tabindex="-1" role="dialog" aria-
labelledby="createPostModalLabel"
      aria-hidden="true">
      <div class="modal-dialog" role="document">
         <div class="modal-content">
            <div class="alert alert-success d-none mb-0" id="successMessage" role="alert">
               <h5 class="alert-heading">
                  Success
               </h5>
               <p class="mb-0">
```

```
                  Your Post was published successfully!
               </p>
            </div>
            <div class="alert alert-danger d-none mb-0" id="errorMessage" role="alert">
               <h5 class="alert-heading">
                  Error
               </h5>
               <p class="mb-0">
                  An error occurred when publishing the Post. Please try again.
               </p>
            </div>
            <div class="modal-header">
               <h5 class="modal-title" id="createPostModalLabel">
                  Create a new Post
               </h5>
               <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                  <span aria-hidden="true">&times;</span>
               </button>
            </div>
            <div class="modal-body p-0">
               <div class="postText" id="postText" contenteditable="true">
               </div>
            </div>
            <div class="modal-footer">
               <button type="button" class="btn btn-outline-secondary" data-dismiss="modal">
                  Close
               </button>
               <button type="button" class="btn btn-outline-success" id="savePostButton">
                  Save Post
               </button>
            </div>
         </div>
      </div>
   </div>

   <!-- Container that contains all the posts -->
   <div class="container postsContainer" id="postsContainer">
   </div>

   <!-- Post template -->
   <div class="container post d-none" id="post" style="margin-bottom: 25px;">
      <div class="card">
         <div class="card-header s4e-postHeader">
            Post #0
         </div>
         <div class="card-body">
            <p class="mb-0">
               Post example
            </p>
         </div>
      </div>
   </div>

   <!-- Back to Top Button -->
   <a id="backToTop" href="#" class="btn btn-lg btn-dark" role="button">Top</a>

   <!-- Footer -->
   <br />

</body>
<script src="frontend/js/speak4env.js"></script>

</html>
```

References

1. https://golang.org/doc/faq

2. https://go.dev

3. https://golang.org/doc/code.html

4. https://tour.golang.org/basics/2

5. https://tour.golang.org/basics/1

6. https://golang.org/doc/effective_go.html

7. https://tour.golang.org/basics/3

8. https://tour.golang.org/basics/4

9. https://tour.golang.org/basics/5

10. https://tour.golang.org/basics/6

11. https://tour.golang.org/basics/7

12. https://tour.golang.org/basics/8

13. https://tour.golang.org/basics/9

14. https://tour.golang.org/basics/10

15. https://tour.golang.org/basics/11

16. https://tour.golang.org/basics/12

17. https://tour.golang.org/basics/13

18. https://tour.golang.org/basics/14

19. https://tour.golang.org/basics/15

20. https://tour.golang.org/basics/16

21. https://tour.golang.org/flowcontrol/1

22. https://tour.golang.org/flowcontrol/2

23. https://tour.golang.org/flowcontrol/3

24. https://tour.golang.org/flowcontrol/4

25. https://gobyexample.com/for

26. https://tour.golang.org/flowcontrol/5

27. https://tour.golang.org/flowcontrol/6

28. https://tour.golang.org/flowcontrol/7

29. https://tour.golang.org/flowcontrol/9

30. https://tour.golang.org/flowcontrol/10

31. https://tour.golang.org/flowcontrol/11

32. https://tour.golang.org/flowcontrol/12

33. https://tour.golang.org/flowcontrol/13

34. https://tour.golang.org/moretypes/1

35. https://tour.golang.org/moretypes/2

36. https://tour.golang.org/moretypes/3

37. https://tour.golang.org/moretypes/4

38. https://tour.golang.org/moretypes/5

39. https://tour.golang.org/moretypes/6

40. https://tour.golang.org/moretypes/7

41. https://tour.golang.org/moretypes/8

42. https://tour.golang.org/moretypes/9

43. https://tour.golang.org/moretypes/10

44. https://tour.golang.org/moretypes/11

45. https://tour.golang.org/moretypes/12

46. https://tour.golang.org/moretypes/13

47. https://tour.golang.org/moretypes/14

48. https://tour.golang.org/moretypes/15

49. https://tour.golang.org/moretypes/16

50. https://tour.golang.org/moretypes/17

51. https://tour.golang.org/moretypes/19

52. https://tour.golang.org/moretypes/20

53. https://tour.golang.org/moretypes/22

54. https://tour.golang.org/moretypes/24

55. https://tour.golang.org/moretypes/25

56. https://tour.golang.org/methods/1

57. https://tour.golang.org/methods/2

58. https://tour.golang.org/methods/3

59. https://tour.golang.org/methods/4

60. https://tour.golang.org/methods/5

61. https://tour.golang.org/methods/6

62. https://tour.golang.org/methods/7

63. https://tour.golang.org/methods/8

64. https://tour.golang.org/methods/9

65. https://tour.golang.org/methods/10

66. https://tour.golang.org/methods/11

67. https://tour.golang.org/methods/12

68. https://tour.golang.org/methods/13

69. https://tour.golang.org/methods/14

70. https://tour.golang.org/methods/15

71. https://tour.golang.org/methods/16

72. https://tour.golang.org/concurrency/1

73. https://medium.com/rungo/anatomy-of-goroutines-in-go-concurrency-in-go-a4cb9272ff88

74. https://tour.golang.org/concurrency/2

75. https://tour.golang.org/concurrency/3

76. https://tour.golang.org/concurrency/4

77. https://tour.golang.org/concurrency/5

78. https://tour.golang.org/concurrency/6

79. https://tour.golang.org/concurrency/9

80. https://github.com/golang/go/wiki/SuccessStories

81. https://github.com/golang/go/wiki/GoUsers

82. https://github.com/avelino/awesome-go

83. Maarten van Steen & Andrew S. Tanenbaum, "Distributed Systems", 2018

84. https://www.w3.org/TR/ws-arch/

85. Μαργαρίτης Κωνσταντίνος, Μιχαηλίδης Παναγιώτης, "Προγραμματισμός Παράλληλων Κατανεμημένων Συστημάτων Υπολογιστών με Java", Πανεπιστημιακές Παραδόσεις, Σχολή Οικονομικών και Κοινωνικών Επιστημών – Τμήμα Εφαρμοσμένης Πληροφορικής και Διοίκησης Τεχνολογίας, 2008

86. https://www.w3.org/TR/mwabp/

87. Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara, "Understanding Web applications through dynamic analysis", Program Comprehension, 2004. Proceedings.

88. https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks

89. Mathias Schwarz, "Design and Analysis of Web Application Frameworks", PhD Dissertation, Aarchus University, 2013

90. Sau Chang Sheong, "Go Web Programming", Manning, Shelter Island, 2016

91. https://github.com/mingrammer/go-web-framework-stars

92. https://go-macaron.com/

93. Changpil Lee, "An Evaluation Model for Application Development Frameworks for Web Applications", Master's Thesis, The Ohio State University, 2012

94. https://github.com/getsentry

95. https://stackoverflow.com/search?q=go+%5Bgo-gin%5D

96. https://stackoverflow.com/search?q=go+%5Bbeego%5D

97. https://stackoverflow.com/search?q=go+%5Bgo-echo%5D

98. https://stackoverflow.com/search?q=go+%5Bbuffalo%5D

99. https://stackoverflow.com/search?q=go+%5Bgo-iris%5D

100. https://gobuffalo.io/en/docs/getting-started/directory-structure